Escola Tècnica Superior d'Enginyeries
Industrial i Aeronàutica de Terrassa

Junio 2011

Estudio e implementación de nuevas funcionalidades de deformación de malla en un software de mecánica de fluidos computacional

MEMORIA

José Plácido Parra Viol

CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 OBJETIVO

El objetivo de este estudio es implementar un tratamiento de malla adecuado para la rotación de un engranaje mediante el uso de un software de dinámica de fluidos computacional de libre distribución: OpenFOAM.

2 JUSTIFICACIÓN

La rotación de un elemento es una aplicación que está actualmente desarrollada en programas comerciales de fluido-dinámica. Sin embargo, el tiempo de las simulaciones excede en algunos casos los límites aceptables. Este motivo lleva a la utilización de varios procesadores simultaneamente, caso en el que se debe de pagar una licencia por cada uno. Con la implementación de un tratamiento de malla adecuado en OpenFOAM será posible llevar a cabo el mismo movimiento y aplicarlo gratuitamente a simulaciones divididas en tantos procesadores como se disponga.

Por otro lado, cabe mencionar que es igualmente interesante poder realizar la simulación en un software de código abierto porque permite entender exactamente qué es lo que se está calculando en cada momento y poder introducir modificaciones. Los softwares comerciales proporcionan directamente resultados sin poder entrar en detalle del proceso aplicado. Actúan, en este sentido, como “cajas negras”.

En definitiva, el objetivo a largo plazo es operar de forma independiente al software comercial por las razones expuestas en este apartado.

3 ALCANCE

Para poder demostrar que un tratamiento de malla es adecuado, es necesario llevar a cabo pruebas en las que se muestre la solución del flujo para comprobar que los resultados tienen coherencia. Este estudio no analiza con más detalle aspectos relacionados con los valores cuantitativos de las variables involucradas en el tratamiento del fluido. Se recurre a ellos únicamente en los casos en los que se pretende validar y/o corroborar que se está manipulando correctamente el movimiento de la malla. Por lo tanto, el esfuerzo se centra en desarrollar todo lo necesario para que se cumpla el objetivo del estudio exclusivamente en términos de malla dinámica.

4 ANTECEDENTES

Dentro de las simulaciones relacionadas con el campo de la fluido-dinámica, destaca el interés tras las casos que involucraban una cierta deformación o movimiento del dominio computacional. Las situaciones ingenieriles en las que hay elementos en movimiento son muy abundantes y por eso es habitual que cualquier distribución de software relacionado con esta rama incluya métodos para tratarlas.

Hoy en día prácticamente todas los fenómenos básicos que incluyen mallas dinámicas son posibles de simular mediante OpenFOAM: rotaciones, desplazamientos lineales, movimiento producido por un fluido... La complejidad aparece cuando se pretende implementar combinaciones de ellos o aplicarlos en un contexto que entraña cierta dificultad. Un ejemplo es la bomba oleohidráulica de engranajes externos que se está investigando numérica y experimentalmente en los departamentos de mecánica de fluidos y aeronáutica de la Escuela Superior de Ingenierías Industrial y Aeronáutica de Terrassa -ETSEIAT- de la Universidad Politécnica de Catalunya -UPC- -[13]-. Es de la empresa Roquet y contiene dos engranajes que encajan uno con el otro para empujar al fluido que circula alrededor de ellos -figura 1-. La rotación de un solo engranaje podría representarse mediante una determinada utilidad de OpenFOAM tal y como se describe más adelante, pero el hecho de que encajen uno con el otro añade ciertas restricciones. En casos como este es posible que se necesite el uso de nuevas funcionalidades de tratamiento de malla.

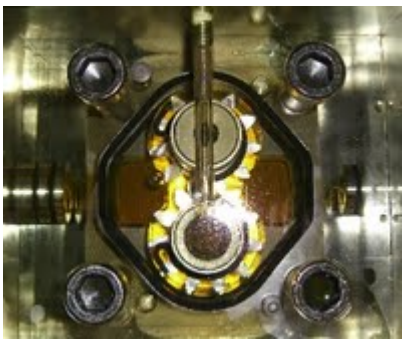


Figura1. Imagen de la bomba oleohidráulica de la empresa Roquet en el laboratorio de mecánica de fluidos de la ETSEIAT. Fuente: [13]

Un ejemplo de una nueva funcionalidad para tratar la rotación de un elemento -una cruz- fue introducida por Liefvendahl y Troëng [3], precisamente mediante el uso de OpenFOAM. Su método propuesto consiste en controlar la calidad de la malla a medida que avanza la simulación. Cuando se detecta un determinado deterioro, un programa -de

propiedad de su centro de trabajo- se encarga de regenerarla automáticamente y reemprender el proceso. Además, el dominio está dividido en dos regiones: una fija en la parte exterior y otra en contacto con el elemento rotativo que es la que sufre las regeneraciones. De esta manera solo se dedica esfuerzo computacional a la zona que se puede ver más afectada por la deformación. Esto es de especial interés por el hecho de que llevaron a cabo las simulaciones con el uso de un solo procesador.

OpenFOAM incluye una opción para tratar geometrías rotativas sin la necesidad de incluir regeneración. Consiste en crear dos mallas. Una exterior y una interior como en el caso de Liefvendahl y Troëng pero esta vez ninguna de ellas sufre deformación. Simplemente la malla interior gira conjuntamente con el elemento rotativo. Para permitirlo es necesario que en ciertos momentos de cada iteración ambas mallas se desvinculen y deslicen entre ellas. Después del movimiento se vuelven a unir dando lugar a una nueva configuración de malla. Desafortunadamente, este método no se puede aplicar a la bomba oleohidráulica mencionada porque el encaje entre engranajes no permite tener una malla alrededor de ellos que haga el papel de malla giratoria.

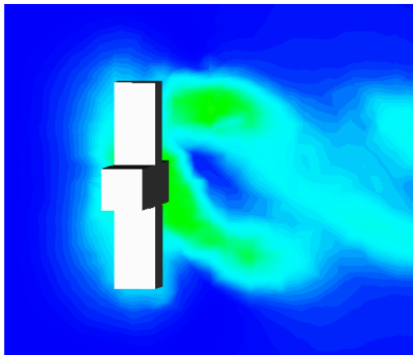


Figura 2. Resultados obtenidos por Liefvendahl y Troëng del flujo a través de una cruz que rota. Fuente: [3].

En definitiva, la situación actual de OpenFOAM respecto a geometrías rotativas obliga a que se tenga que desarrollar algún tipo de funcionalidad para que se puede implementar el movimiento rotativo en las condiciones de la turbo-bomba mencionada. Mientras que , por otro lado, conocidos paquetes comerciales como por ejemplo *Ansys Fluent* si que permiten la simulación de casos complejos como los mencionados engranajes. Este tipo de software es muy completo y abarca un enorme rango de posibilidades. De hecho, la investigación de la bomba oleohidráulica en la ETSEIAT -[13]- se está llevando a cabo con *Fluent* a falta de tener implementado aún el mismo movimiento en OpenFOAM.

5 INTRODUCCIÓN A OPENFOAM

5.1 ¿QUÉ ES OPENFOAM?

OpenFOAM es un paquete de software de libre divulgación que utiliza diferentes métodos de discretización para resolver ecuaciones en derivadas parciales. Aunque es básicamente conocido por su uso en dinámica de fluidos, también abarca campos como las finanzas, el electromagnetismo o el análisis tensional de sólidos. Se trata de una librería C++ a partir de la cual se crean ejecutables, conocidos como aplicaciones. Existen dos tipos de aplicaciones: *solvers* y utilidades. Los *solvers* se ocupan de resolver las ecuaciones que gobiernan un fenómeno físico. Las utilidades son aplicaciones que se utilizan antes, o bien después de haber empleado el solver. Se utilizan para manipular datos con el objetivo de preparar el problema para ser tratado por el *solver* y permitir un cómodo y correcto análisis de los resultados. Un esquema de la organización global de OpenFOAM se muestra en la figura 3.

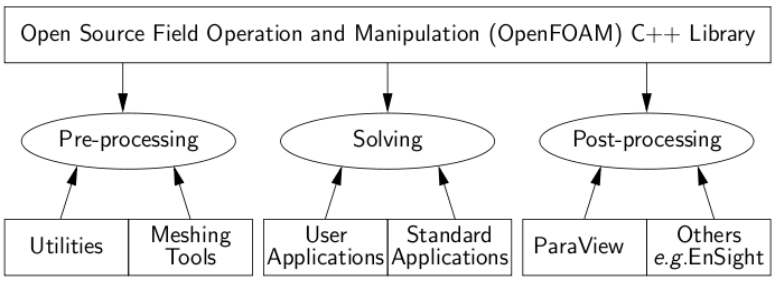


Figura3. Esctructura general de OpenFOAM según las herramientas utilizadas en cada paso de una simulación completa.

Por lo tanto, el esquema general de la forma de operar con OpenFOAM a nivel de usuario consistirá en realizar tareas de pre-proceso mediante utilidades, resolver con el uso de *solvers* y post-procesar con utilidades de nuevo. En función del tipo de problema que se desee analizar, se escogerán el tipo de utilidades y *solvers* que sean coherentes y adecuados para el fenómeno a estudiar des de el punto de vista práctico. Sin embargo, puede que no exista ningún *solver* que permita solucionar la situación en cuestión. En este caso, OpenFOAM permite la libre ampliación y modificación por parte de sus usuarios de cualquier tipo de aplicación, hecho para el que es necesario un cierto conocimiento tanto de programación en C++ como de la forma en la que el software está organizado.

5.2 CONCEPTOS BÁSICOS SOBRE EL CÓDIGO

Desde el punto de vista de la programación en C++, OpenFOAM está estructurado en un gran número de clases. Es lo que se conoce como programación orientada a los objetos. Mediante esta forma de estructurar el código se crean ciertos elementos que contienen variables y las correspondientes funciones que permiten acceder a ellas. Dichos elementos reciben el nombre de clases, y se pueden entender como si fueran tipos de variables personalizadas en las que se puede imponerr cúantas y qué tipo de variables posee y de qué forma se van a manipular. A continuación se muestra un ejemplo de definición de clase para ilustrar mejor el concepto.

```
class rectangulo {
    int base, altura;
  public:
    void pon_valores (int,int);
    int da_area ()
};
```

Código 1. Ejemplo de declaración de una clase.

En el código anterior, la primera línea declara una clase -objeto- llamada *rectangulo*. A continuación se definen las variables que posee dicho objeto: dos números enteros llamados *base* y *altura*. En la tercera línea, la palabra *public* informa de que lo que va a aparecer a continuación serán las variables o funciones que se podrán acceder y que podrán modificar *base* y *altura*. Según el código mostrado, la única forma de modificar las variables será mediante las funciones *pon_valores* y *da_area*. La función *pon_valores* introduce el valor de *base* y *altura* y la función *da_area* devuelve un entero que corresponde a la multiplicación de *base* por *altura.*

Las clases son el parámetro clave de OpenFOAM desde el punto de vista de la programación. Todo el código está basado en relaciones entre clases mediante lo que se conoce como amistad y herencia. Una clase *amiga* de otra puede acceder a su dominio privado, que en el ejemplo anterior serían las variables *base* y *altura*. Por otro lado, una clase *heredera* posee en su dominio privado algún aspecto que es común a la clase de la cual hereda. Gracias a estas características el código de OpenFOAM evita duplicaciones y con la relación entre clases aparece una jerarquía que ayuda a entender mejor su estructura.

Además, para organizar mejor la estructura del programa OpenFOAM utiliza lo que se cononce como *namespaces*. Son como unos contenedores, o espacios, que agrupan funciones y clases bajo un mismo nombre que permite categorizar los elementos del

programa. Hay decenas de *namespaces* y cada uno se nombra de forma que proporciona una idea del tipo de elementos que se encuentran en su interior. El *namespace* global que incluye al resto recibe el nombre de *Foam*, y para acceder a otro espacio a partir de éste se utiliza el operador *::*. Por ejemplo, para acceder al espacio *meshTools*, que agrupa funciones y variables utilizadas para hacer modificaciones simples en la malla, el código sería *Foam::meshTools*.

Mediante el uso de *namespaces* se pueden utilizar en una misma función variables con el mismo nombre siempre que provengan de diferentes espacios. De esta manera se evitan errores de redefinición y permite manejar los elementos de forma más conceptual gracias a que el nombre del espacio del que provienen ayuda a entender qué se está manipulando exactamente.

5.3 ESTRUCTURA DE LOS CASOS

Un caso es como se denomina a un conjunto de carpetas y archivos que definen un problema ingenieril específico y la forma en que se va a resolver -el *solver* que se va a utilizar-. Para poder solucionar un fenómeno se deben de crear tres carpetas: c*onstant, system* y una carpeta temporal inicial. Esta última no tiene porque corresponder con el tiempo 0 ya que el caso puede ser una parte de un proceso que haya simulado cierto tiempo previamente.

La carpeta *constant* contiene una carpeta llamada *polyMesh*, donde se encuentra toda la información acerca de la malla, y archivos que son necesarios para el solver proporcionándole información como por ejemplo propiedades físicas del fluido. Dependiendo del tipo de solver y del tipo de problema a tratar se necesitará una mayor o menor cantidad de archivos auxiliares.

En *system* se incluyen archivos relacionados con el proceso de resolución. Como mínimo hay tres: *controlDict*, *fvSchemes*, y *fvSolution*. El primero establece parámetros relacionados con la ejecución como por ejemplo el tiempo de comienzo, el incremento de tiempo o cada cuántas iteraciones se desea guardar resultados. El segundo se ocupa de elegir cuáles van a ser los esquemas númericos utilizados para resolver las ecuaciones que aparecen en el solver. Por ejemplo, si hay que realizar una interpolación, se puede elegir si se desea que sea lineal, cúbica, centrada... El archivo *fvSolution* determina que esquema de solucionador de ecuaciones va a utilizar para cada una de las variables a calcular. Entre las opciones aparecen famosos algoritmos de resolución de esquemas numéricos como el Gauss-Seidel.

Para poder llevar a cabo la simulación, cada caso debe tener una carpeta temporal inicial en la que se establezcan las condiciones iniciales de las variables de interés. Dentro de la carpeta temporal habrá tantos archivos como variables vayan a ser calculadas durante la simulación. Por ejemplo, en el caso de un flujo incompresible, las variables serían la velocidad y la presión, por lo que la primera carpeta temporal contendrá dos archivos fijando los valores de éstas variables en el instante inicial. De todos modos, en el caso de que la variable sea un vector -como en el caso de la velocidad-, se puede definir su estado mediante un archivo para cada una de sus componentes, definidas como valores escalares. En definitiva se trata de definir las condiciones iniciales y de contorno del problema en cuestión.

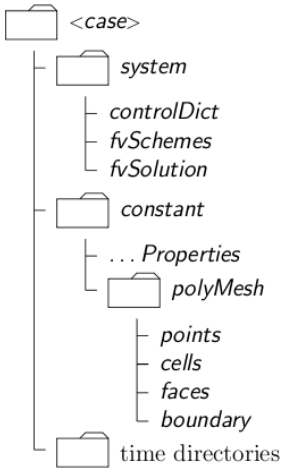


Figura 4. Estructura de un caso en OpenFOAM.

Las tres carpetas citadas son las necesarias para poder ejecutar una simulación en OpenFOAM. Una vez empieza el proceso de cálculo, para cada incremento temporal existirán unos nuevos valores de las variables iniciales. Estos resultados serán guardados en carpetas temporales que tendran como nombre el instante temporal al que pertenecen. El contenido de las carpetas de resultados varía según el tipo de problema tratado y el solver utilizado. Normalmente habrá archivos que contendrán los nuevos valores de las variables y una carpeta llamada *uniform* donde hay un archivo que indica el instante al que pertenecen los valores. Sin embargo, si la malla se desplaza, aparecerá una carpeta llamada *polyMesh* que contendrá los nuevos puntos de la malla, así que el contenido de las carpetas temporales depende del tipo de simulación. En la figura 5 se muestra un caso antes y después de haber ejecutado la simulación.

Cabe mencionar que la carpeta temporal inicial no tiene porque corresponder siempre al instante inicial cero. La simulación puede estar dividida en varias partes y entonces el tiempo inicial de una parte será el final de la anterior. En este caso la carpeta inicial

temporal deberá nombrarse según corresponda y el cambio deberá reflejarse en el *controlDict* dentro de *system* para que todo funcione correctamente. De lo contrario, OpenFOAM buscará la carpeta inicial 0 para empezar la simulación y aparecerá un error puesto que no la encontrará.

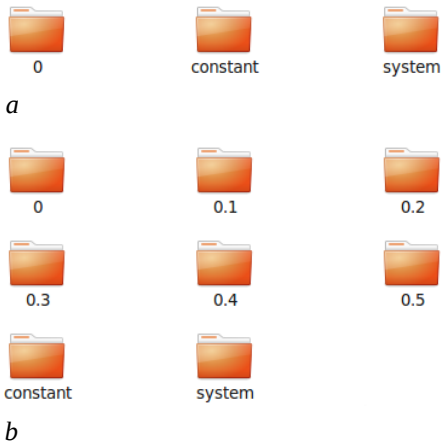


Figura 5. Apariencia de un caso en OpenFOAM. En *a* antes de la simulació; en *b* después.

5.4 REALIZACIÓN DE UNA SIMULACIÓN

Para realizar una simulación se debe crear de forma adecuada el caso, es decir, las carpetas y archivos necesarios que se han mencinado en el apartado 5.3. Para llevar a cabo este proceso primero se debe de crear la malla y los diferentes archivos que necesita el *solver* para funcionar. Se puede crear un caso entero partiendo desde cero pero es mucho más recomendable copiar en una nueva carpeta un tutorial ya existente y a partir de ahí ir modificando según se desee.

Existen tres formas diferentes de crear una malla en la versión 1.7.1 de OpenFOAM: mediante la utilidad *blockMesh*, con *snappyHexMesh e* importándola de otro software .

5.4.1 CREACIÓN DE MALLA CON *BLOCKMESH*

BlockMesh crea una malla a partir de un archivo que se llama *blockMeshDict*, que se podría traducir como el diccionario de *blockMesh*. Como indica el nombre, la malla se creará a partir de bloques que pueden tener diferentes formas geométricas (hexaedros, prismas, tetraedros, pirámides...). Se trata de una utilidad que deja de ser práctica en

cuanto la geometría a realizar es complicada.

A continuación se detalla cada uno de los pasos que se deben seguir para generar una malla con esta utilidad analizando los elementos que tiene un *blockMeshdict* vacío como el que se muestra a continuación. El código hasta la primera línea de asteriscos azules es una cabecera común para todos los *blockMeshDict* existentes. El diccionario se modifica a partir de esa línea delimitadora.

```
/*--------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2010 OpenCFD Ltd.
     \\/     M anipulation  |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
vertices
(

);
blocks
(

);
edges
(
);
patches
(

);
mergePatchPairs
(
);
// ************************************************************************* //
```

Código 2. Código de un *blockMeshDict* sin rellenar.

En primer lugar, dentro de los paréntesis que hay debajo de la palabra *vertices* se determinan los puntos de la malla que se van a crear mediante sus tres coordenadas espaciales. Si la malla es un cuadrado, se deberán introducir cuatro vértices

correspondientes a cada una de las esquinas. La forma de escribir las coordenadas es la siguiente: -x y z-, donde x, y, z representan las coordenades en cada uno de los ejes homónimos. No importa donde se fije el sistema de referencia mientras se mantenga la coherencia.

La palabra *vertices* denota el comienzo de un subdiccinoario. Hay varios archivos de diferentes utilidades que acaban su nombre en *Dict -dynamicMeshDict* o *mapFieldsDict* por ejemplo- y todos ellos están formados por lo que se llaman *subDict*, o subdiccionarios. En el caso de *blockMeshDict*, está formado por cinco subdiccionarios llamados *vertices, blocks, edges, patches* y *mergePatchPairs*.

El segundo paso es definir los bloques a partir de los vértices introducidos. Esto se lleva a cabo dentro del subdiccionario *blocks*. En función del tipo de bloque que queramos que forme nuestra malla se utilizará una palabra clave u otra, pero la sintaxis para definir un bloque es la siguiente:

```
geom (0 3 1 ...) (ncx ncy ncz) simpleGrading (gx gy gz)
```

Código 3. Declaración de un bloque en *blockMeshDict*.

En *geom* se ha de poner la palabra clave para la geometría deseada del bloque, las posibles opciones se detallan en la guia del usuario -[11]- aunque la más común es *hex* -hexaedro-.

En el primer paréntesis se han de colocar los vértices que formarán el bloque separados por espacios. Los puntos suspensivos indican que en función del tipo de bloque el número de vértices puede variar, por ejemplo en el caso del hexaedro se necesitan seis vértices para definirlo. Es importante tener en cuenta que no se deben introducir las coordenadas de los vértices, sino su número. Para OpenFOAM el primer vértice definido en el subdiccionario *vertices* es el número cero, el siguiente el uno y así sucesivamente. La única parte en la que se introducen coordenadas es en el subdiccionario *vertices*. El resto de subdiccionarios se basan simplemente en la númeracion de éstos con la excepción de los bordes curvados como se verá más adelante.

En el segundo paréntesis se ha de poner en *ncx* el número de celdas que se desean en esa dirección y lo mismo con los otros dos valores pero en las dos direcciones restantes. En definitiva se decide cuantos puntos intermedios se van a crear entre los vertices en cada una de las direcciones. Los puntos que se crearán serán el número de celdas que se desee menos 1, teniendo en cuenta que el número de celdas ha de ser siempre un número

entero.

La palabra *simpleGrading* determina el tipo de expansión de las celdas y el paréntesis que le sigue el valor de dicha expansión. En *gx* se colocara cúanto mayor se desea que sea la última celda en la dirección x respecto a la primera. Por ejemplo, si *gx* es la unidad, no habrá variación; si es dos, el tamaño de las celdas en dirección x irá aumentando linealmente hasta que en la última celda el tamaño sea el doble del inicial. De la misma manera se puede modificar el tamaño de las celdas en las otras dos direcciones con *gy* y *gz.*

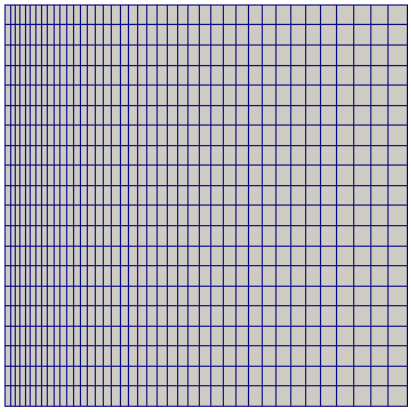


Figura 6. Malla con expansión de celdas en dirección horizontal.

Existe otra palabra clave para la expansión de celdas: *edgeGrading.* En este caso se puede elegir la expansión de cada uno de los bordes del dominio por separado.

El tercer paso consiste en rellenar el subdiccionario *edges*, que se utiliza en caso de que se deseen crear geometrías curvas. Permite definir los bordes a partir de unos determinados puntos de interpolación. Lo que se ha definido hasta el momento en el *blockMeshDict* genera por defecto líneas rectas si no se introduce nada en este subdiccionario. Hay diferentes palabras claves como en el caso de los bloques, por ejemplo la palabra *arc*. A continuación se muestra un ejemplo para crear un arco de circumferencia entre los vértices 1 y 5, pasando por el punto de coordenadas 1.1 0.0 0.5.

```
Arc 1 5 (1.1 0.0 0.5)
```

Código 4. Declaración de un borde curvado en *blockMeshDict*.

En cuarto lugar se define lo que se llaman *patches.* Un *patch* es una definición que se aplica a una o más áreas de una geometría que no tienen por qué estar físicamente conectadas. Por ejemplo, las partes superior e inferior de un cubo pueden definirse

conjuntamente en un *patch*. Este elemento aparece por el hecho de que la malla no solo tiene funciones geométricas sino que también va a jugar un papel importante en la simulación a través de las condiciones de frontera. El tipo de *patch* que se adjudica a la frontera del dominio es importante porque determinará que tipo de condiciones de frontera podrán imponerse sobre el. En definitiva, cada pared del dominio que deba de realizar una función diferente -moverse, permanecer quieta, hacer de entrada/salida de flujo...- se definirá en un *patch* diferente. De esta manera, la frontera de cualquier dominio queda dividida en uno o más patches -habitualmente más de uno-.

En la sección 5.2 de la guía del usuario -[11]- se mencionan los tipos de *patch* existentes, sin embargo, más adelante se detallarán algunos de los tipos más comunes. La forma de definirlo queda ilustrada en el siguiente ejemplo:

```
wall movingWall

        (
                        (1 5 13 1)
                        (5 6 14 13)
        )
```

Código 5. Ejemplo de definición de un *patch* de tipo *wall* en *blockMeshDict*.

En el código anterior la palabra *wall* es un tipo de *patch*. *MovingWall* es el nombre inventado para ese *patch*, formado por dos caras -superfícies- de la malla, definidas cada una por cuatro vértices. Es importante notar que se utilizan cuatro vértices para el ejemplo utilizado porque las paredes a incluir en el *patch* tienen cuatro lados y vértices. Si se tratara de una cara triangular únicamente haría falta indicar los tres vértices que pertenecen a esa cara. Además, los números no indican coordenadas, sino el orden al que pertenecen los vértices, tal y como se ha indicado anteriormente.

El último subdiccionario se utiliza en caso de que se pretenda realizar algún tipo especial de unión entre bloques. En el caso de que se quiera una malla en que cada uno de los bloques se une con el adyacente compartiendo los puntos del contorno no se debe incluir nada en esta parte del archivo.
Una vez finalizado el *blockMeshDict*, la forma de crear la malla es accediendo mediante el terminal -OpenFOAM funciona en Linux- a la carpeta del caso y ejecutar el comando *blockMesh.* Entonces aparecerán dentro de la carpeta *constant/polyMesh* cuatro archivos adicionales con datos como por ejemplo la lista de todos los puntos de la malla -no solo los vértices- y sus coordenadas.

En este momento cabe mencionar que todas las malla de OpenFOAM son

tridimensionales, por eso los vértices se definen con tres coordenadas. De todos modos se puede tratar un caso como si fuera bidimensional o unidimensional, e incluso axisimétrico, mediante las condiciones de frontera de las paredes del dominio. Lo único que hay que hacer en el caso bidimensional o unidimensional es que los *patch* de las paredes que no interesen sean de tipo *empty*, es decir vacío. OpenFOAM adjudica estas propiedades a las fronteras del dominio que no esten incluidas en ningún *patch* definido por el usuario. En el caso axisimétrico las condiciones de frontera a utilizar para los patches es *wedge*. En definitiva, con estas condiciones de frontera se fuerza a que la solución no varíe en una , o varias, direcciones y obtener así un análisis personalizado.

5.4.2 SNAPPYHEXMESH E IMPORTACIÓN

La creación de mallas mediante *snappyhexmesh* y la importación no están desarrolados con tanta profundidad como el caso de *blockMesh* porque la experiencia con ellos es mínima y son, por tanto, menos conocidos.

SnappyHexMesh parte de una imagen tridimensional ya creada que debe de estar en formato *Stereolithography* -STL- y la coloca en una malla base que la engloba, normalmente creada con blockMesh por su sencillez. En pocas palabras, el proceso consiste en adaptar la malla a la figura STL paso a paso. Como en el caso de blockMesh el proceso consiste en escribir un diccionario que en esta ocasión recibe el nombre de snappyHexMeshDict mediante el cual se determinan las acciones a realizar. El objetivo es refinar la malla base en las zonas en contacto con el cuerpo modelado en formato STL y después eliminar las celdas que se encuentran total o parcialmente dentro del cuerpo. Por último, la malla se engancha al cuerpo para cubrir los espacios que se generan al eliminar las celdas interiores.

En referencia a la importación de mallas de otro software, OpenFOAM dispone de varias utilidades que permiten a los usuarios importar mallas creadas con otro software como Fluent, Gambit, Star-CD...

5.4.3.SELECCIÓN DE SOLVER

El *solver* a utilizar depende del tipo de problema físico que se pretenda resolver y sus características: incompresible o compresible, estacionario o transitorio, laminar o turbulento... En la sección 3.5 de la guía del usuario -[11]- aparece una lista de todos los *solvers* existentes en la versión 1.7.1. De esta manera, la elección del *solver* no debe ser

una dificultad porque OpenFOAM facilita información sobre las caracterísiticas de cada uno de ellos. Lo que si entraña una cierta dificultad es incluir en el caso todos los archivos que el *solver* necesita consultar para funcionar correctamente. Es bastante complicado deducir cuáles deben incluirse o eliminarse a través de un análisis del código por lo que el método más sencillo consiste en fijarse en los tutoriales, que están agrupados en función del *solver* que utilizan para simular la solución.
La mayoría de *solvers* necesitan que se incluya algún archivo en el caso. Como se ha mencionado anteriormente, el archivo deberá colocarse dentro de la carpeta *constant.* Los archivos típicos a añadir se refieren normalmente a propiedades físicas del fluido, de manera que adaptarlo al caso específico de que se trate es tan directo como cambiar los correspondientes valores. Sin embargo, hay ocasiones en las que el archivo a añadir no tiene nada que ver con propiedades, como por ejemplo el caso en que la malla se desplaza. Este hecho necesita que se incluya un diccionario llamado *dynamicMeshDict* en el que se determinará la forma de tratar dicho movimiento por parte del *solver.*

En referencia a la preparación de la carpeta *system* y la temporal, es bastante más rápida y sencilla. En *system* se debe modificar el *controlDict* para elegir el incremento temporal, el tiempo de finalización y el intérvalo de guardado de resultados principalmente. Hay otros parámetros tales como precisiones y formatos de datos que se pueden modificar pero en general en el *controlDict* se modifican básicamente los aspectos relacionados con la ejecución temporal de la simulación. A continuación nos hemos de asegurar que en *fvSchemes* están incluidos y definidos los esquemas numéricos de todas las operaciones que realice el *solver* y que en *fvSolution* se incluya el tipo de solucionador de ecuaciones para cada una de las variables que pretendemos calcular durante la simulación.

Por último, en la carpeta temporal inicial se deben de establecer los valores de todas las variables que se van a calcular durante la simulación, incluido el movimiento de la malla si procede. Dentro de la carpeta se ha de incluir un archivo para cada variable en el cual se establecerán una seríe de parámetros que son esenciales para el comienzo de la ejecución. Al principio se especifican las unidades de la variable y a continuación se establece el valor de la variable tanto para la malla interna como para el contorno. En este punto entran en juego los *patch* que se han mencionado con anterioridad, ya que se podrán prescribir tantas condiciones de frontera como *patchs* existan. A cada *patch* se le puede imponer una condición diferente y de ahí la razón de que en una misma frontera existan normalmente varios *patch*, que evita que toda la frontera se comporte de forma similar.

En función del tipo de clase al que pertenecen las variables OpenFOAM incluye multitud de condiciones de contorno a aplicar -incluso más de cuarenta-. Estos cambios están

sobretodo motivados por la diferencia entre las variables de tipo vector y las escalares. Entre las opciones disponibles se puede imponer un valor fijo, oscilante, dependiente del tiempo, calculado según otros parámetros de la simulación ...etc. A continuación se incluye un trozo de código que impone un valor fijo nulo en el *patch* llamado *fixedWalls*.

```
fixedWalls
            {
                type            fixedValue;
                value           uniform (0 0 0);
            }
```

Código 5. Ejemplo de imposición de condición de frontera. A la velocidad en el *patch fixedWalls* se le aplica un valor fijo de 0.

Por el valor impuesto en el *patch fixedWalls* se puede deducir que se trata de una variable de tipo vector por el hecho de introducir tres componentes, por ejemplo, sería válido para la velocidad del fluido. Los vectores en OpenFOAM deben siempre estar localizados entre paréntesis, de lo contrario el programa solo leería el primer número de los tres considerándolo un escalar y proporcionaría un error.

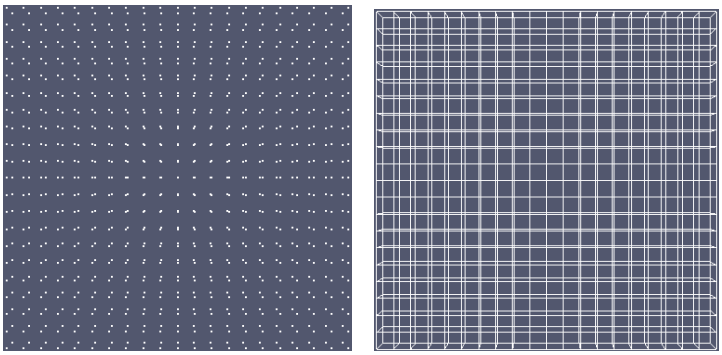
5.4.4. EJECUCIÓN Y POSTPROCESADO

Para ejecturar la simulación en OpenFOAM se utiliza el terminal de comandos. Mediante él se debe acceder a la carpeta que contenga el caso y, mediante ciertos comandos, ejecutar las utilidades y *solvers* adecuados. Por ejemplo, para un caso en el que la malla se genere mediante *blockMesh*, una vez dentro de la carpeta del caso se debe escribir *blockMesh* en el terminal para crear la malla. Si la malla es importada o creada con *snappyHexMesh* el proceso es diferente, pero en todo caso involucra el uso del terminal para ejecutar las debidas utilidades.

Una vez la malla ya está preparada el siguiente paso es solucionar el caso con el *solver*, también mediante un comando en el terminal que coincide con el nombre del *solver*. Hay ocasiones en las que el proceso no es tan directo porque antes se deben ejecutar utilidades que acaban de preparar las variables o realizan interpolaciones de otras simulaciones.

Una vez se ha realizado la simulación, dentro de la carpeta del caso habrán aparecido nuevas carpetas temporales con los valores calculados de las variables. Para visualizar los resultados OpenFOAM incluye una utilidad llamada *paraFoam* que utiliza el

software *paraView*, aunque también se puede postprocesar mediante otros softwares tanto comerciales como de libre distribución. A lo largo del desarrollo del estudio se ha utilizado *paraView* por el hecho de que ya está incluido en el paquete de OpenFOAM y es sencillo de manejar.



a *b*

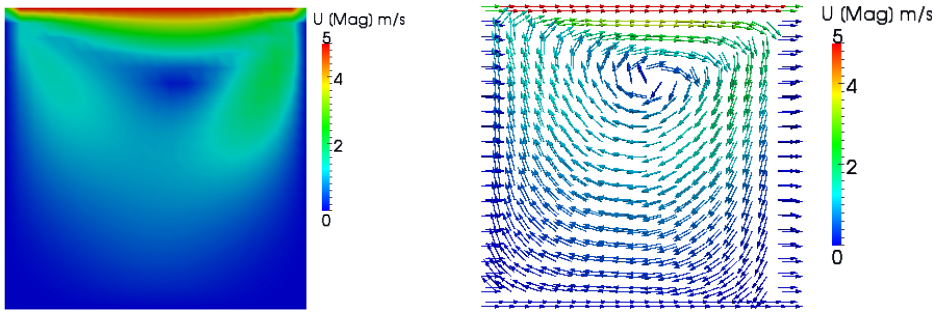
Figura 7. Diferentes formas de mostrar una malla en *paraView*. a: puntos; b: entramado de puntos.

Al escribir el comando *paraFoam* desde el terminal se abre *paraView* con los resultados ya leídos. Es la única parte de la simulación en la que existe una interfaz gráfica para el usuario, el resto de tareas se realizan mediante el terminal. *ParaView* se utiliza para visualizar tanto la malla creada como los resultados obtenidos. Se puede abrir antes de realizar la simulación numérica simplemente para comprobar que la malla que se ha generado es la deseada, pudiéndose comprobar cada *patch* por separado. No hace falta abrir ningún archivo porque al ejecutar el comando *paraFoam* el programa ya carga todos los datos existentes. Hay que tener en cuenta que solo se pueden visualizar los instantes en los que se ha guardado una carpeta temporal. Mediante *controlDict* se puede variar el número de carpetas de resultados que se van a crear y, además, si hay restricciones de espacio también se puede indicar que los datos se guarden en formato comprimido.

De las numerosas opciones que ofrece *paraView* para observar los resultados las más prácticas y destacadas son tres principalmente: visualizar la malla, visualizar resultados en código de colores o la opción anterior con flechas que indican la dirección. Al cargarse la interfaz gráfica y el caso, lo primero que aparece es el dominio de la malla. A partir de ahí se puede elegir si se prefiere ver los puntos, la malla en sí y algunas opciones más -figura 7-.

En relación a los resultados, la forma más rápida de observarlos es mediante una escala

que tiene asignada un color a cada magnitud de la variable mostrada. De esta manera se pueden observar las zonas de mayor o menor presión y/o velocidad para obtener una idea de como se comporta el flujo en esa simulación. Sin embargo, a veces es importante comprobar que la dirección del flujo es la adecuada y en este caso se pueden incluir flechas para facilitar la interpretación. Además, las flechas estarán coloreadas de acuerdo a la misma escala que en el caso de no incluirlas, aunque óbviamente no tiene sentido utilizar este aspecto para escalares como la presión. Por tanto, se puede obtener, en una sola imagen, información sobre la magnitud y la dirección de una variable de una forma sencilla y rápida. En la figura 8 se muestran dos ejemplos de visualización de la velocidad en *paraView*.



a *b*

Figura 8. Formas de representar un vector. a: solo magnitud en código de colores; b: magnitud en código de colores y dirección con flechas.

6 DINÁMICA DE FLUIDOS COMPUTACIONAL -CFD-

OpenFOAM es utilizado principalmente en el campo de la dinámica de fluidos computacional -Computational Fluid Dynamics-. Un hecho que lo demuestra es la gran cantidad de *solvers* de fluidos que contiene en comparación con los dedicados a otros campos como el electromagnetismo o el análisis de esfuerzos en sólidos. CFD se ha convertido hoy en día en una de las principales herramientas utilizadas en la investigación de la mecánica de fluidos y gracias a OpenFOAM está al alcance de todos.

6.1 ORIGEN DE CFD

Las ecuaciones que gobiernan la mecánica de fluidos surgen básicamente de establecer la conservación de la masa y la cantidad de movimiento en un volumen de control que corresponede con una región de interés para el fenómeno de estudio. Si se tratara de un sólido se podría escoger una masa en su lugar -masa de control- pero en el caso de los fluidos es más complicado seguir una porción de masa que pasa rápidamente por la región de interés. En el análisis del volumen de control aparecen las llamadas ecuaciones de Navier-Stokes, conocidas ya desde hace aproximadamente un siglo. Desde sus inicios, su utilidad ha estado limitada a la solución de un reducido número de tipos de flujo durante muchos años. Dichas soluciones servían para entender los fundamentos de la mecánica de fluidos pero rara vez encontraban una aplicación directa de análisis o diseño ingenieril, hecho que forzaba la necesidad de utilizar datos empíricos en la mayoría de ocasiones. En ese momento quedaron demostradas las dificultades de llevar a cabo ensayos reales con resultados válidos: condiciones que son prácticamente imposibles de reproducir con modelos a escala, aparatos de medida que influyen demasiado en el flujo, dificultad añadida por geometrías complejas... De modo que se observó que los ensayos empíricos eran prácticos para obtener aspectos globales como por ejemplo resistencias, sustentaciones o caídas de presión. Para analizar situaciones complejas los medios requeridos pueden aumentar demasiado el coste o el ensayo puede implicar una inaceptable cantidad de tiempo.

La gran alternativa apareció gracias a los ordenadores. A pesar de que las bases necesarias para la solución de las ecuaciones con métodos numéricos gozaban también de cierta antigüedad, estas no fueron muy útiles hasta el surgimiento de las computadoras. Desde entonces, la evolución de la capacidad de cálculo y el espacio de almacenamiento han sido espectaculares y todavía no muestran signos de frenarse. Este

hecho provocó un gran aumento del interés en los métodos numéricos que hizo el estudio de la mecánica de fluidos más fácil y efectivo. Muchos de los investigadores del campo se vieron atraídos por la nueva posibilidad que se ha convertido hoy en día en una propia rama conocida como la mecánica de fluidos computacional -Computational Fluid Dynamics-.

6.2 MÉTODOS NUMÉRICOS

Las ecuaciones de conservación de cantidad de movimiento y masa son en la mayoría de casos complicadas. Son no lineales, acopladas y dificiles de resolver, hecho que hace que no sea posible garantizar mediante las herramientas matemáticas actuales la existencia de una solución única dadas unas ciertas condiciones de contorno. Una solución analítica solo es posible de obtener en casos muy básicos que, como se ha mencionado, carecen de relevancia. Incluso en casos donde se hayan realizado ciertas simplificaciones -hecho muy común en los casos en los que existe solución- la resolución es compleja y lo habitual es recurrir a los métodos numéricos.

Para llegar a la solución mediante métodos numéricos se debe de realizar una discretización del dominio cuya calidad influye determinantemente en la validez de los resultados obtenidos, del mismo modo que en un ensayo empírico estos dependen de la calidad de las herramientas utilizadas. Ahora no se obtiene la solución para el volumen de control completo, sino para ciertos lugares discretos de éste en el espacio y en el tiempo.

No se debe olvidar que las soluciones numéricas son siempre aproximaciones debido a la existencia de varias fuentes de errores. Primero, el propio proceso de discretización de las ecuaciones es aproximativo. En segundo lugar, las ecuaciones diferenciales utilizadas están normalmente modificadas para que la solución sea posible. Seguramente se habrán despreciado y/o anulado términos en función de las hipótesis aplicables al problema en cuestión que también dejan su huella en la exactitud de los cálculos. Además, a la hora de resolver el problema se utilizan métodos iterativos que para que reproduzcan la solución exacta se necesitarían simulaciones de duración inaceptable. Se puede conseguir reducir el error de la discretización mediante unas aproximaciones más precisas o aplicando éstas sobre una región más pequeña. Sin embargo, esto repercutiría negativamente en el coste y el tiempo necesario para llevar a cabo el cálculo de modo que se llega a una situación de compromiso.

Los métodos numéricos son una opción más para obtener soluciones a un cierto

fenómeno físico. Por el hecho de no estar absentos de errores, como todos los demás métodos, sus resultados deben de ser exhaustivamente analizados antes de llegar a conclusiones. Por muy atractiva y coloreada que aparezca una solución no significa nada a no ser que lo que exprese tenga sentido y este dentro de unos márgenes de tolerancia.
Es interesante entender todas y cada una de las partes que forman un método de solución numérica para poder asimilar conceptos más avanzados que puedan aparecer más adelante en este estudio. Se presenta a continuación un breve resumen de los elementos principales y esenciales.

- Modelo matemático: Es el punto de partida. Es la representación matemática completa del fenómeno que se va a intentar resolver mediante los esquemas numéricos. Está formado por las ecuaciones en derivadas parciales y las condiciones de contorno y puede que ya se la haya aplicado algún tipo de simplificación como las que se mencionan en el apartado 6.3. Normalmente, se identifica de antemano la forma en la que se va a solucionar el modelo ya que la manera de enfocar el proceso puede depender de cuál sea el resultado deseado.
- Método de discretización: Permite dividir el dominio continuo en un cierto número de puntos de cálculo. Es decir, debe de aproximar el modelo matemático de forma que produzca resultados en posiciones discretas del tiempo y el espacio. El resultado es, como mínimo, una ecuación similar a la del modelo matemático para cada uno de los puntos discretos.
- Sistema de coordenadas: Se debe definir el tipo de sistema de coordenadas que se van a utilizar -cartesianas, polares...- para poder expresar el modelo numérico en función de este último.
- Malla: La malla está formada por el conjunto de lugares discretos en los que se calcula el valor de las variables. Es básicamente una representación discreta del dominio completo.
- Aproximaciones finitas: Son la forma en la que se van a aproximar las derivadas e integrales que aparecen en las ecuaciones del método de discretización. Una vez aplicadas, las ecuaciones se convierten en simples expresiones algebraicas.
- Método de solución: Existen varias formas de abordar un esquema numérico. Se debe de establecer un cierto proceso que permita ir calculando los valores de las variables para cada uno de los puntos del domino de la forma más eficiente y simple posible. Un ejemplo de método de solución sería un esquema iterativo en el que los valores iniciales son asumidos y a partir de ahí se itera hasta conseguir la convergencia. El abanico de posibilidades es amplio: puede que se incluyan factores de relajación, que no se realicen iteraciones -no muy común-...
- Criterio de convergencia: Finalmente, se debe de establecer una cierta condición

que haga que el proceso finalice. Se puede, por ejemplo, calcular la diferencia entre la iteración anterior y la actual para establecer un límite a partir del cual ya no se repita el proceso.

Los métodos numéricos son herramientas matemáticas que se utilizan básicamente para llegar de una forma más sencilla a una solución que puede ser muy compleja. Como esquemas matemáticos, deben de cumplir ciertos criterios que aseguren que la solución va a ser coherente y válida. Para empezar, la discretización debe ser tal que cuando se reduce el espaciado temporal y/o geométrico entre mallas la ecuación discretizada coincida con la ecuación exacta. La diferencia entre la ecuación exacta y la discretizada recibe el nombre de error de truncamiento y debe de ser nula en condiciones en las que el incremento temporal y/o la distancia entro nodos -puntos de cálculo- tienden a 0.

Además, el método no debe de divergir. Esto lo consigue un método que no incrementa los errores existentes a medida que el proceso iterativo avanza. Cumplir este criterio puede requerir que el incremento temporal sea menor que un cierto valor o que se deban de utilizar factores de relajación.

Por otro lado, cuando el espaciado de los nodos tiende a 0 la solución de la ecuación discretizada debe tender a la solución de la ecuación exacta.

El esquema debe de satisfacer las leyes de conservación. Es decir, la cantidad de una variable conservada que entra en un volumen debe de ser la misma que la sale.

Los resultados provenientes del esquema numérico deben de encontrarse dentro de su rango de valores posibles. Por ejemplo, un porcentaje debe de estar entre el 0% y el 100%, o una cantidad que siempre es positiva -por ejemplo la densidad- no puede aparecer con signo negativo como resultado de una manipulación de un método numérico. Si se trata de un modelo que pretende reproducir un tipo de fenómeno muy complejo, hay que asegurarse que los resultados producidos son realistas.

6.3 MODELOS SIMPLIFICADOS

Debido a la complejidad de las ecuaciones de Navier-Stokes, en los casos en los que existe solución muchos términos son nulos. Además, se puede observar que para diversos tipos de flujo, unos son más relevantes que otros y se pueden descartar los de menos influencia. Existen varios tipos de flujo muy conocidos por el hecho de que permiten realizar importantes simplificaciones a las ecuaciones. Son los siguientes:

- Flujo incompresible: Permite asumir que la densidad del fluido no varía. Además, en el caso de ser isotérmico tampoco lo hace la viscosidad.
- Flujo de Euler o invíscido: Tiene lugar lejos de elementos sólidos, donde se puede suponer que los efectos de la viscosidad son despreciables. En este caso las ecuaciones de Navier-Stokes se reducen a las ecuaciones de Euler -de ahí el nombre-.
- Flujo potencial: Es uno de los más simples. Se trata de un flujo de Euler añadiendo la característica de que el rotacional de la velocidad es nulo. En este caso se puede asumir que la velocidad proviene de un potencial. Es importante su entendimiento sobretodo a nivel académico pero carece de practicidad.
- Flujo de Stokes: Se da en situaciones en las que la velocidad del fluido es muy pequeña, la viscosidad muy elevada o las dimensiones geométricas minúsculas. Al predominar los términos viscosos el número de Reynolds se reduce y los términos inerciales de las ecuaciones de Navier-Stokes se pueden eliminar. Además, si se cree oportuno, se pueden eliminar los términos no-estacionarios por las bajas velocidades.
- Hipótesis de Boussinesq: Se aplican a un flujo que incluye transferencia de calor en el que sus propiedades son función de la temperatura. Consiste en suponer que la densidad es constante excepto en el término gravitacional. Cabe mencionar que esta simplificación puede producir errores o resultados irreales si la variación de temperatura es mayor de un cierto margen -15º para el aire según [1]-.
- Hipótesis de la capa límite: Este tipo de flujo mezcla una parte invíscida -fuera de la capa límite- y otra en la que la fuerza dominante es la viscosidad -dentro de la capa límite-.
- Modelos de fenómenos complejos: Hay casos en los que es imposible describir matemáticamente el comportamiento del fluido. Las turbulencias, la combustión o flujos multi-fase son ejemplos de este tipo de situaciones. Para su representación matemática se requiere el uso de modelos semi-empíricos que deben ser estudiados con mucho detalle ya que pueden afectar a la precisión global del resultado con sus errores.

6.4 EL ANÁLISIS MEDIANTE VOLÚMENES FINITOS

Como se ha mencionado, OpenFOAM utiliza sobretodo métodos de discretización basados en volúmenes finitos para solucionar las ecuaciones, lo que consiste en dividir la zona en la que se pretende estudiar el comportamiento del fluido en pequeños

volúmenes. Cada una de estas pequeñas celdas va dar lugar a una ecuación que deriva de la aplicación de los debidos principios de conservación. El lugar en el que se resuelve cada ecuación recibe el nombre de nodo y normalmente está situado en el centro geométrico del volumen al que corresponde. Por tanto, el sistema de ecuaciones a resolver tendrá tantas ecuaciones como celdas, lo que significa que cuanto más pequeños sean estos volúmenes mayor será el esfuerzo computacional necesario para resolver el dominio completo.

El valor de las variables se debe de determinar también en puntos distintos del centro de las celdas. Por ejemplo, se necesitan valores en las caras de los volúmenes para calcular el flujo de alguna propiedad a través de ellas. La forma de obtenerlos es mediante la interpolación entre los valores de los nodos. Aunque existen varias formas de realizar este último paso, el método más utilizado es la interpolación lineal, que ha sido la utilizada en todas las simulaciones realizadas en este estudio.

En los volúmenes que se encuentran en la frontera del dominio el tratamiento del valor de las variables en las caras es diferente. Como en el exterior de la frontera no hay nodos, la información se debe obtener de las condiciones de frontera. Por ejemplo, en el caso de tratarse de la parte de la frontera que se utiliza como entrada de fluido, se debe de establecer un cierto flujo o velocidad según los intereses del análisis. Gracias a estos datos se completa la resolución del dominio sin añadir ninguna ecuación.

7 LA MALLA DINÁMICA

7.1 TIPOS DE MALLAS

Mediante el uso de los métodos numéricos la solución se obtiene para ciertos puntos concretos del dominio seleccionado. El usuario es el responsable de seleccionar el número de puntos en los que se desea realizar el cálculo y debe tener en cuenta la considerable variación del tiempo de ejecución que conlleva un aumento o disminución de estos. No tiene sentido ejecutar un cálculo con un gran número de puntos si lo único que interesa es realizar una primera comprobación y no un análisis de los resultados, la simulación tardaría demasiado tiempo y no valdría la pena.

Todos los puntos en los que se van a realizar los cálculos forman lo que se conoce como malla. Cuando se habla de malla dinámica se hace referencia al movimiento de los puntos mencionados por el hecho de que hay alguna parte de la geometría que se desplaza o deforma a medida que avanza la simulación.

Es muy común que los puntos de una malla se muestren unidos siguiendo un cierto critero de manera que se cree un entramado que tiene la apariencia de una rejilla -de ahí el nombre de malla-. Según las características geométricas de dicho entramado se pueden clasificar las mallas en estructuradas y no estructuradas.

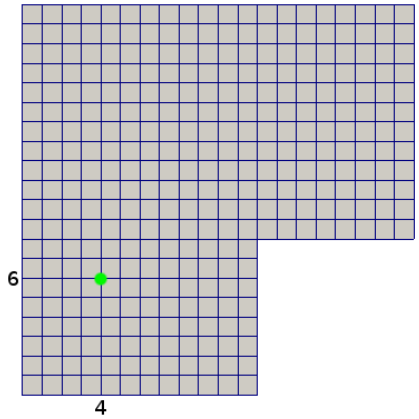


Figura 9. Ejemplo de malla estructurada. El punto verde representa la coordenada 4-6, única en el dominio.

En una malla estructurada los puntos siguen un cierto orden. Se divide el dominio en varias partes tanto en horizontal como en vertical, formando una especie de cuadrícula como si se tratara de coordenadas cartesianas. Para formar el entramado partiendo, por

ejemplo, del tercer punto de la división horizontal, se debe avanzar a través de las divisiones verticales uniendo todos los puntos que correspondan a la tervera división horizontal. Partiendo de cualquier punto de la frontera y trazando una línea en dirección perpendicular a la frontera, se debe de cumplir que esta línea no sea cruzada por ninguna otra que parta de la misma frontera. De modo que cada punto del dominio se puede expresar mediante coordenadas como se muestra en la figura 9.

Este tipo de mallas son simples y es recomendable su aplicación en casos en que la geometría del dominio sea senzilla. El principal problema de este tipo de entramado es que en caso de desear una mayor concentración de puntos en algún lugar del dominio por motivos de precisión, la malla sería también fina en otras partes del dominio en las que la resolución no es tan relevante, conllevando un desperdicio de recursos.

La malla estructurada puede estar definida mediante bloques de forma que el dominio quede dividido en varias regiones con diferente densidad de puntos. La flexibilidad es la principal ventaja de este método porque permite refinar únicamente en las zonas en las que interesa, podiendo reducir la cantidad de puntos en las zonas de menor interés -ver figura 10-.

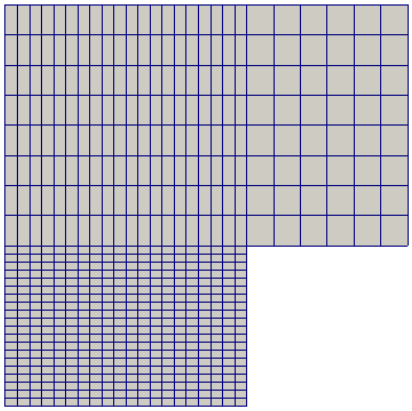


Figura 10. Ejemplo de malla estructurada formada por tres bloques diferentes.

Existe la opción de que varios bloques estén superpuestos. No es un caso muy común pero tiene aplicación por ejemplo en situaciones en las que se debe de seguir el movimiento de un objeto. Uno de los bloques se desplaza con el elemento mientras que el otro permanece inmóvil cubriendo sus alrededores.

Por otro lado, las mallas no estructuradas no siguen ningún tipo de patrón. No existe una división ordenada del dominio y los puntos están unidos entre ellos de forma arbitraria. Sin duda se trata de la mejor opción cuando la geometría es muy compleja, situación en

la que la creación de una malla estructurada puede resultar más complicada. No existe ninguna restricción ni en la forma de las celdas ni en la cantidad de puntos vecinos de cálculo.

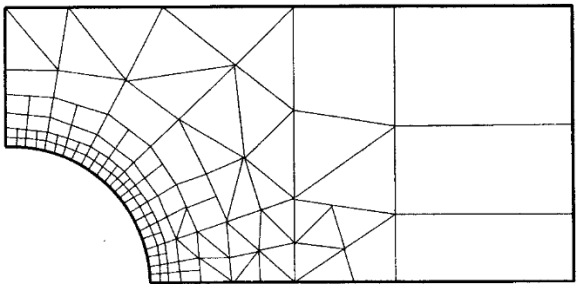


Figura 11. Ejemplo de malla no estructurada. Fuente: [1].

Desde el punto de vista negativo, cabe mencionar que este tipo de mallas dan lugar a matrices algebraicas más irregulares y sin estructura diagonal que conllevan un cierto aumento en el tiempo de cómputo. Además eso hace que los términos sean más complicados de tratar e identificar dentro del algoritmo.

7.2 MOVIMIENTO DE MALLA

Son muchos los casos en los que el movimiento de alguna superficie u objeto produce una deformación del dominio que obliga a los puntos de la malla a reestructurarse. Algunos ejemplos son el movimiento de los actuadores de un ala, movimiento lineal de pistones, movimiento rotativo de componentes de turbo-maquinária o hélices o el movimiento libre de un cuerpo bajo la influencia de un fluido. La acción en estos casos va más allá de controlar la geometría inicial de la malla. Se debe tratar de forma robusta la deformación de los puntos causada por el movimieto de alguna de las fronteras para asegurar que los resultados producidos siguen siendo igual de válidos que en el caso sin desplazamiento. Por lo tanto, es necesario plantear un sistema de ecuaciones adicional que añade una cierta complicación a la simulación.

El punto de partida es el movimiento de las fronteras del dominio. Se puede considerar conocido ya sea porque está predefinido o porque forma parte de la solución. En el caso de un desplazamiento prescrito se sabe exactamente cuál va a ser el comportamiento de las fronteras a lo largo de toda la simulación. Por otro lado, si la causa del movimiento es la influencia del flujo en el objeto, éste será proporcionado por la propia solución. Por ejemplo, si se pretende simular la reacción de un cuerpo flotante ante el impacto de una onda -ver figura 12-, el movimiento durante la simulación no está predefinido por el

usuario. Existe una dependencia con la acción provocada por el fluido. Sin embargo, la parte más importante es el papel de los puntos internos del dominio.

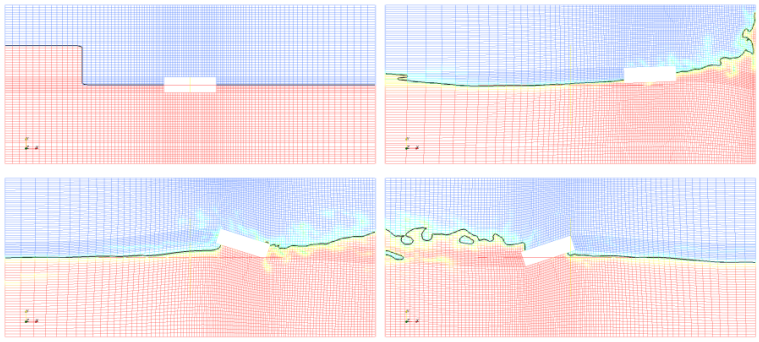


Figura 12. Ejemplo de simulación en la que el movimiento de la frontera es parte de la simulación. Fuente: [6].

Cuando alguna frontera del dominio se desplaza o deforma, la malla ha de reaccionar de forma que se mantenga su validez. Por esta razón ha de tener lugar una determinada deformación de sus puntos de manera que se adapte lo mejor posible a lo que está ocurriendo. Aquí es donde aparece el verdadero trabajo del tratamiento de la malla dinámica. Se debe de establecer un *solver* que se dedique a calcular el desplazamiento de todos y cada uno de los puntos de la malla para cada instante de la simulación. La malla resultante debe de mantener la robustez original dentro de lo posible para reducir al mínimo los posibles errores debidos a su cambio de geometría. Obviamente el efecto en el esfuerzo computacional de la simulación se verá incrementado, sobretodo cuando se traten mallas muy densas o deformaciones exigentes.

El movimiento que experimentan los puntos de la malla es completamente independiente de la solución del flujo. No importa si se trata de un caso incompresible, compresible, con o sin transferencia de calor... El *solver* que manipula la malla no debe en ningún caso adaptarse a las condiciones del fluido ni viceversa. Se debe elegir el correcto tratamiento para cada una de las dos partes por separado.

7.3 TRATAMIENTO DEL MOVIMIENTO DE LA MALLA

Existen dos formas generales de manipular mallas mediante OpenFOAM. La más senzilla es la que se ocupa únicamente de calcular el movimiento de los puntos internos para reaccionar lo mejor posible al desplazamiento de alguna frontera. Recibe el nombre de movimiento de malla automático, o simplemente deformación de malla. Deberá incluirse siempre que haya una deformación o desplazamiento del contorno del dominio.

Por otro lado, hay una forma más compleja de manipular mallas en las que se modifica su topología. A diferencia del cálculo de los puntos internos, los cambios topológicos modifican el número de elementos de la malla o su conectividad, por ejemplo, abriendo o cerrando un conducto durante la simulación. Veamos con más detalle las características de las dos técnicas.

7.3.1 MOVIMIENTO AUTOMÁTICO DE MALLA

La deformación de malla es el más simple de los casos y es el que se ha aplicado en el estudio. Consiste en mover los puntos para que se adapten al cambio de forma -o movimiento- de alguna de las fronteras del dominio y de esa forma preservar la calidad y validez de la malla. El objetivo es que los puntos sigan en posiciones bien distribuidas del dominio para evitar introducir errores relacionados con la discretización.

De las varias posibilidades existentes para tratar este tipo de movimiento, las primeras en utilizarse fueron las basadas en métodos de volúmenes finitos -FVM-. Sin embargo, este planteamiento obtiene soluciones para el centro de las celdas y los puntos que interesan son en realidad los vértices. Este tratamiento requiere realizar interpolaciones que comportan un coste computacional adicional y, además, se detectó una gran dificultad para evitar la degeneración de celdas. Por estas razones esta aproximación al movimiento de la malla fue dejada de lado y los esfuerzos se centraron en desarrollar un método basado en los vértices de los pequeños volúmenes en los que se divide el dominio.

Uno de las alternativas propuestas recibe el nombre de analogía de muelles -*spring analogy*-. El objetivo es tratar las líneas entre vértices como muelles y obtener el movimiento de los puntos mediante las cargas que se comunican desde los puntos situados en las fronteras. Las ecuaciones que gobiernan el comportamiento de los puntos provienen de plantear un equilibrio de fuerzas en cada uno de ellos. Es un sistema lineal, lo que significa que es muy ligero computacionalmente.

Desafortunadamente, con esta configuración existían varios errores que forzaron a añadir ciertas complicaciones. La ley de rigidez debía ser exponencial y se tuvieron que añadir muelles torsionales a cada uno de los puntos. De modo que las ecuaciones de equilibrio de fuerzas pasaron a ser no lineales y aparecieron unas ecuaciones adicionales de equilibrio de momentos. Finalmente, lo que parecía que iba a ser un método muy sencillo y rápido se convirtió en una torpe aproximación.

Al mismo tiempo, se había conseguido solucionar de forma fiable el movimiento de

malla mediante un método de elementos finito -FEM- basado en la ecuación de Laplace a un coste computacional bastante más reducido. Esto hizo que se volcara todo el esfuerzo en el estudio de este tipo de métodos como la mejor alternativa para tratar la deformación de malla. En concreto se utilizaba la ecuación de Laplace con difusividad variable, donde la difusividad interpretaba de alguna manera el mismo papel que la rigidez en el método de la analogía de muelles.

OpenFOAM, aunque se basa prácticamente de forma exclusiva en los métodos de volúmenes finitos, utiliza un *solver* con esquema FEM que parte de la división en tetrahedros de las celdas polihédricas que aparecen en la discretización mediante FVM. El *solver* ofrece una decena de posibilidades de tratar la difusividad variable, entre ellas la difusividad constante. En la imagen se muestra el mismo movimiento de malla tratado con difusividad constante, direccional e inversamente proporcional a la distancia a la frontera en movimiento. Se puede observar la gran influencia que tiene la elección de la difusividad sobre la deformación de la malla y su calidad.

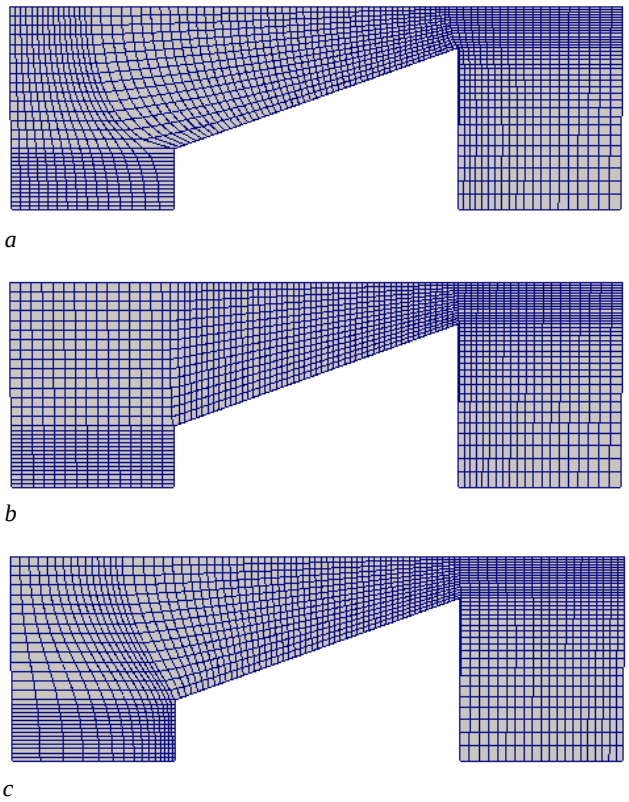


Figura 13. Movimiento de puntos según el tipo de difusividad escogido. a: constante; b: direccional; c: inversamente proporcional a la frontera en movimiento.

7.3.2 CAMBIOS TOPOLÓGICOS

Si existen movimientos muy amplios de superficies, puede que no sea suficiente utilizar el movimiento de puntos. Grandes deformaciones llevan a degradaciones en la calidad de la malla que requieren la introducción de los cambios topológicos. Por ejemplo, en el caso de un componente rotativo, si dejamos que los puntos se vayan deformando a medida que se efectúa el giro, se observará como en poco tiempo la malla se convierte en inválida -figura 14-. Algunos puntos conseguirían tales deformaciones que la malla se vería obligada a adoptar formas muy radicales e incluso a invadir el espacio fuera de la propia frontera del campo. En estas condiciones una malla con conectividad entre puntos fija no soportaría más giro o el error debido a la incorrecta distribución de los puntos sería muy grande. Otro ejemplo sería un caso en el que una frontera se aproxima o aleja demasiado produciendo una densidad de puntos excesiva o insuficiente. Para que en estas condiciones extremas se pueda mantener la validez y calidad de la simulación se debe modificar algo más que la geometría de la malla.

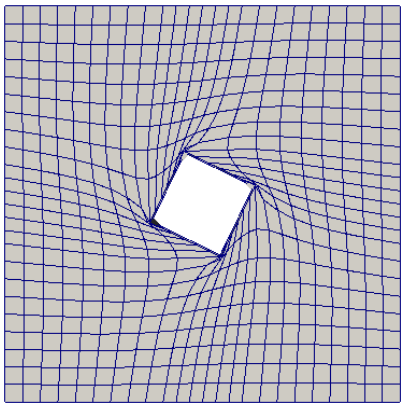


Figura 14. Excesiva deformación que convierte la malla en inválida.

Un cambio topológico produce variaciones en la malla que implican alteraciones del número de elementos -puntos, caras, celdas- o de la conectividad entre ellos. Los cambios topológicos están basados en tres principales actuaciones: añadir, quitar o modificar una celda, cara o punto. Mediante la combinación de estas tres opciones, conocidas como cambios primitivos, se pueden construir complejas manipulaciones de malla que dan una mayor libertad respecto al movimiento automático, en el que los puntos simplemente se desplazan. Utilizarlos de forma independiente en función de las necesidades resulta bastante complejo y laborioso, por lo que lo habitual es que estos cambios se engloben dentro de algoritmos más complejos denominados modificadores

topológicos.

Un modificador topológico es un código encargado detectar las condiciones en las que se ha de lanzar una cierta acción y ejecutarla. Se establece antes de empezar la simulación y no es necesaria ninguna actualización durante el transcurso del cálculo. A continuación se describen los que se incluyen en la distribución de OpenFOAM:

- Incluir-eliminar frontera: Permite que un determinado grupo de caras del dominio se conviertan en fronteras a las que, por lo tanto, se les podrán aplicar algún tipo de condición de contorno. Una aplicación típica de este modificador es la apertura o cierre de una válvula a través de la que circula fluido. La señal que hace que se ejecute su código suele estar relacionada con el tiempo de ejecución, por ejemplo: válvula abierta 3s; cerrada 10s y repetir el proceso.
- Adición-substacción de celda: Como su nombre indica, elminina o añade celdas al dominio en función de sus dimensiones. Se establece un espesor mínimo y máximo de forma que el modificador elimina o añade, respectivamente, celdas cuando estos valores se alcanzan. Es una aplicación muy utilizada en movimientos que producen grandes expansiones o compresiones de los puntos internos.
- Interfaz deslizante: Consiste en desvincular dos partes de la malla y permitir que una deslize sobre la otra entre instantes temporales. Al final de cada iteración, ambas se vuelven a unir dando lugar a una nueva conectividad. Gracias a este modificador, zonas de la malla pueden desplazarse ampliamente respecto a otras sin conllevar deformaciones excesivas en los puntos internos.

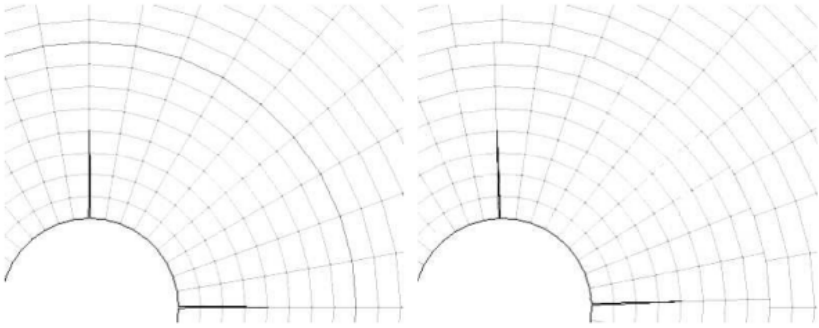


Figura 15. Ejemplo de movimiento de malla con interfaz deslizante. Fuente: [6].

Según las propiedades de los modificadores topológicos mencionados, la interfaz deslizante podría aplicarse al caso de rotación del obstáculo. Para ello, la zona de la malla que rodea al elemento debería desvincularse del resto para permitir la rotación.

Entonces, en cada instante del proceso de cálculo la conectividad entre ellas sería distinta pero no existiría deformación en ninguna de las dos partes. Cabe decir que se trata, en realidad, de una alternativa factible y aplicable.

Como se ha mencionado en el objetivo del estudio, la finalidad es aplicar el movimiento a un engranaje fabricado. Desafortunadamente, el engranaje está diseñado para que encaje con otro idéntico -figura 16-, de manera que no se pueden dejar unas zonas de malla a sus alrededores porque se invadirían una a la otra. Por lo tanto, ningún modificador de topología parece ser adecuado para el estudio, por lo que se centra la atención en buscar nuevas posibilidades dentro del movimiento automático.

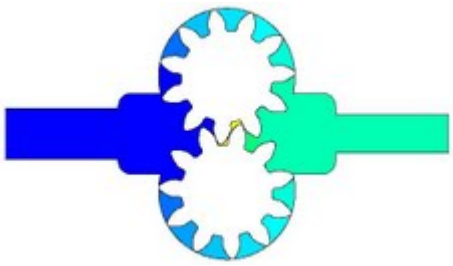


Figura 16. Bomba oleohidráulica de la empresa Roquet modelizada en Fluent. Se muestran los engranajes para ilustrar el encaje. Fuente: [13].

7.4 INDICADORES DE CALIDAD DE MALLA

Cuando hay deformación de los puntos internos se debe tener en cuenta que esto va a tener cierta repercusión en las propiedades de la malla. Si el movimiento de las fronteras es muy amplio puede que se alcancen geometrías complejas que deterioren su capacidad. Por suerte, cuando ocurre algún error en OpenFOAM la simulación se detiene. Así, se puede analizar más detenidamente la situación de los puntos en instantes previos al del fallo. Sin embargo, interesa tener una cierta valoración global de la calidad de la malla aunque no se haya llegado a una situación extrema. La utilidad *checkMesh* realiza esta tarea e indica si los valores están dentro de los márgenes aceptables. Los principales parámetros indicadores de la calidad de una malla son: no-ortogonalidad, oblicuedad y la relación de areas.

- No-ortogonalidad - θ_{no} -. Es el ángulo entre S_f y d_f según la figura 17. Cuando el plano de la cara es perpendicular -ortogonal- al vector que une los centros de las celdas este parámetro es nulo. Indica de alguna forma el grado de inclinación que está sufriendo la cara. Según el manual OpenFOAM advanced 1.6 se pueden

tener las siguientes situaciones:

- $\theta_{no} < 50^{o}$: Rango que no necesita corrección. Aceptable.
- $50^{o} < \theta_{no} < 70^{o}$: Se necesita una corrección limitada.
- $70^{o}\,\theta_{no} < 80^{o}$: Es posible que se pueda continuar con la simulación pero la precisión se verá comprometida.
- $\theta_{no} > 80^{o}$: Muy complicado que se pueda continuar con la simulación.

- Oblicuedad: Proporciona de forma cuantitativa la proximidad entre la intersección de d_f con el plano de la cara y el punto f. Si la cara contiene lados muy largos en comparación con el resto, el punto f puede estar situado en un lugar más alejado del centroide.

- Relación de áreas: Da una idea de lo "cuadrada" que es una celda. Se trata de una relación entre área y volumen que da una idea de la distorsión que puede estar sufriendo dicho elemento.

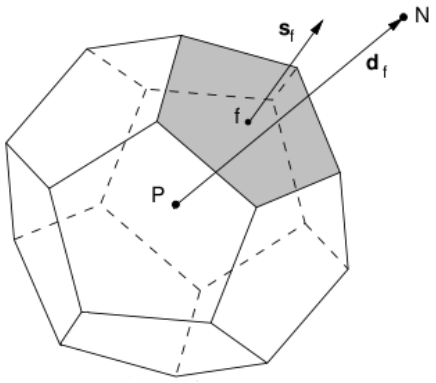


Figura 17. Celda polihédrica donde: P es el nodo computacional, f el centroide de la área coloreada, S_f un vector perpendicular al área, N el nodo computacional de una celda vecina y d_f el vector entre P y N. Fuente: [6].

8 DESARROLLO DE UNA NUEVA FUNCIONALIDAD

Para alcanzar el objetivo del estudio se plantea la evolución en dos etapas: conseguir la rotación de un elemento y lograr que se mantenga la calidad de la malla durante dicha rotación. A continuación se detallan los pasos tomados, enfocados a ir avanzando fase a fase.

8.1 MODIFICACIÓN DE LA GEOMETRÍA DE UN CASO

Las modificaciones de la geometría forman parte de las consideradas a nivel de usuario. Incluyen cambios que no requieren la variación de ningún código interno de OpenFOAM. Se llevan a cabo principalmente a través de la modificación de los "diccionarios", es decir, los archivos que acaban en Dict, aunque también requiere la manipulación de archivos de propiedades y condiciones de contorno. Un caso se puede modificar por completo a nivel de usuario ya que permite determinar tanto la geometría como las propiedades del fluido.

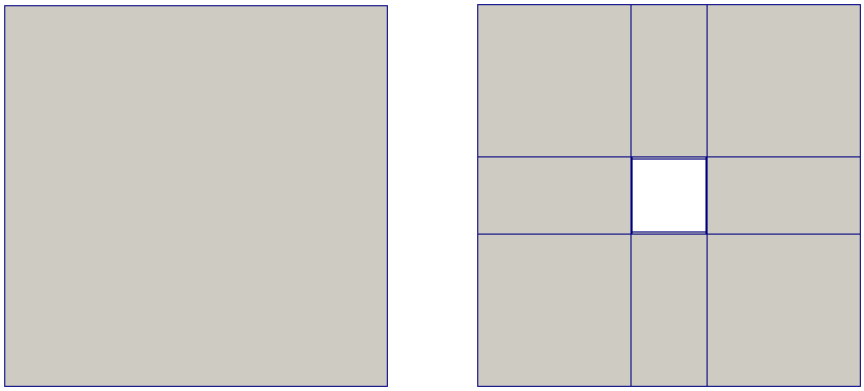
8.1.1 EL TUTORIAL *CAVITY* CON OBSTÁCULO

El primer paso que se ha llevado a cabo en este estudio es la modificación de la geometría del tutorial llamado *Cavity* añadiendo un cuadrado en el centro del dominio. Aprovechando por completo el caso original, el único archivo que hay que utilizar es el *blockMeshDict*. Como se ha mencionado en el apartado 5.4.1, aquí se debe de definir la geometría de la malla mediante bloques -hexaedros-. En el caso original solo existe un bloque ya que el campo completo es un hexaedro, pero en el caso del obstáculo habrá que crear la malla a partir de más bloques. Para su definición se deben de añadir no solo las coordenadas de los vértices del obstáculo sino también sus proyecciones sobre la frontera.

A partir de las nuevas coordenadas se divide el dominio en ocho bloques como muestra la figura 18. Cuatro de ellos tienen una cara en contacto con el obstáculo y los otros cuatro tienen un borde en contacto con el obstáculo. El problema que surge con esta distribución es que es difícil cuadrar el número de celdas de cada bloque para que la distribución sea equitativa. Sin embargo, probando con varias opciones se puede llegar rápidamente a una en que prácticamente no se note el cambio, aunque no se trata de un

factor muy relevante.

La modificación realizada afecta únicamente a la malla, pero los cambios introducidos han creado una nueva frontera. Todas las propiedades deben ser definidas en la superfície añadida para que la simulación se pueda realizar correctamente. Esto se lleva a cabo en los archivos *p* y *U* dentro de la carpeta temporal inicial, tomando como ejemplo la declaración de las fronteras preexistentes.



a *b*

Figura 18. Distribución de bloques en el caso *cavity*. a: caso original; b: con obstáculo añadido.

Es muy importante tener en cuenta el orden en que se nombran los vértices cuando se definen los bloques. Existe un patrón publicado en la guía del usuario de OpenFOAM que indica cuál es la secuencia que se debe seguir. Aún sin segur dicha referencia es posible que la malla se cree, pero, en consecuencia, las líneas que unen los puntos se pueden entrecruzar y no sería posible ejecutar la simulación. En cualquier caso, el papel de *paraView* en la visualización de la malla es clave para comprobar que el resultado es el deseado.

Cabe decir que se ha creado un obstáculo cuadrado por su sencillez. El dominio resultante es geométricamente muy simple y la malla utilizada poco densa para que permita hacer simulaciones rápidas. Si se consigue rotar el cuadrado, para hacer lo mismo con el engranaje lo único que hay que hacer es cambiar la geometría. El resto de parámetros -condiciones de frontera, *solvers* del fluido y del movimiento de malla...- no necesitan ninguna modificación.

8.2 CREACIÓN DE UN SOLVER INCOMPRESIBLE CON TRATAMIENTO DE MALLA

El estudio pretende crear un *solver* propio capaz de tratar malla dinámica. Por su simplicidad, se ha decidido que esté diseñado para tratar un flujo incompresible. En versiones anteriores de OpenFOAM existe un *solver* incompresible para malla dinámica llamado *icoDyMFoam*, sin embargo, en la versión utilizada para el estudio -1.7.1- éste no existe. Por lo tanto, se debe crear una nueva versión del antiguo solver, que es uno de los primeros pasos prácticos de este estudio.

Se puede tratar de crear cualquier cosa en OpenFOAM desde cero, pero eso requiere un alto conocimiento de la estructura del programa y mucha experiencia en su uso, a parte de un avanzado nivel en la programación en C++. La forma más práctica y rápida de crear un nuevo solver es partiendo de uno ya existente. En el caso de este estudio, se partirá del *solver* incompresible original llamado *icoFoam*. Además, como se desea incluir cambios para tratar el movimiento de la malla se utilizará también como referencia un *solver* de malla dinámica llamado *pimpleDyMFoam*. *IcoFoam* se aprovechará al completo ya que lo único que hace falta es añadir código en relación con la malla. Por otro lado, *pimpleDyMFoam* se estudiará para localizar los aspectos necesarios para poder tener una malla en movimiento.

El objetivo es reconocer, dentro de *pimpleDyMFoam* cuales son las partes del código que permiten que la malla dinámica se mueva y actualice cada incremento de tiempo. Hay algún *solver* más que también incluye malla dinámina, sin embargo, se ha elegido éste como referencia porque existe un solver llamado *pimpleFoam*, que es exactamente el mismo pero sin cambios de malla. Así, comparando los dos *solvers*, que son equivalentes excepto en el aspecto de que uno es capaz de solucionar movimientos de malla, será más fácil identificar cuáles son las variaciones que se deben incorporar a *icoFoam* para convertirlo en un solucionador de mallas que se deforman.

Sin embargo, el proceso no termina aquí. Tal como ocurre en los casos, se deben incluir diferentes archivos que el *solver* debe de utilizar para funcionar. Entonces, una vez ya se haya incluido el código para tratar la malla dinámica en *icoFoam*, el siguiente paso será incluir todos los archivos necesarios, con las correspondientes modificaciones, para que el *solver* pueda acceder a ellos y funcione de forma correcta y adecuada.

8.2.1 INTERPRETACIÓN DE PIMPLEDYMFOAM

Este solver es muy similar al que se pretende crear -icoDyMFoam- en varios aspectos. Desde el punto de vista del tipo de fluido que tratan ambos son incompresibles y además los dos solucionan flujos transitorios. La diferencia entre ellos reside en el hecho de que pimpleDyMFoam trata la turbulencia y utiliza un proceso diferente para solucionar las ecuaciones: el proceso pimple, que es una mezcla de dos procesos llamados piso y simple.

El objetivo es identificar las partes de este *solver* que tratan el movimiento de la malla. Con este fin se realiza, en primer lugar, un análisis del código del *solver pimpleDyMFoam*. A continuación se muestra el código de dicha aplicación del cual se realizará una descripción básica. En segundo lugar, se comparará el código con el *solver pimpleFoam*, que es equivalente pero no incluye el desplazamiento de la malla. De este modo se podrán identificar las partes que se deben trasladar al original *icoFoam* para convertirlo en *icoDyMFoam.*

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2010 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Application
    pimpleDyMFoam.C
Description
    Transient solver for incompressible, flow of Newtonian fluids
    on a moving mesh using the PIMPLE (merged PISO-SIMPLE) algorithm.
   Turbulence modelling is generic, i.e. laminar, RAS or LES may be selected.

\*---------------------------------------------------------------------------*/
1       #include "fvCFD.H"
```

```
2       #include "singlePhaseTransportModel.H"
3       #include "turbulenceModel.H"
4       #include "dynamicFvMesh.H"
5
6   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
7
8       int main(int argc, char *argv[])
9       {
10          #include "setRootCase.H"
11          #include "createTime.H"
12
13           #include "createDynamicFvMesh.H"
14
15            #include "readPIMPLEControls.H"
16            #include "initContinuityErrs.H"
17            #include "createFields.H"
18          #include "readTimeControls.H"
19
20     // * * * * * * * * *  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
21
22      Info<< "\nStarting time loop\n" << endl;
23
24  while (runTime.run())
25   {
26       #include "readControls.H" //añadido
27       #include "CourantNo.H"
28
29       // Make the fluxes absolute
30       fvc::makeAbsolute(phi, U);
31
32        #include "setDeltaT.H"
33
34        runTime++;
35
36        Info<< "Time = " << runTime.timeName() << nl << endl;
37      //Empieza añadido
38        mesh.update();
39
40        if (mesh.changing() && correctPhi)
41        {
42            #include "correctPhi.H"
43        }
44
45        // Make the fluxes relative to the mesh motion
46        fvc::makeRelative(phi, U);
47
48        if (mesh.changing() && checkMeshCourantNo)
49        {
50            #include "meshCourantNo.H"
51        }
52      //Acaba añadido
53
54        // --- PIMPLE loop
```

```
55        for (int ocorr=0; ocorr<nOuterCorr; ocorr++)
56        {
57            if (nOuterCorr != 1)
58            {
59                p.storePrevIter();
60            }
61
62            #include "UEqn.H"
63
64    //empieza añadido
65            // --- PISO loop
66            for (int corr=0; corr<nCorr; corr++)
67            {
68                rAU = 1.0/UEqn.A();
69
70                U = rAU*UEqn.H();
71                phi = (fvc::interpolate(U) & mesh.Sf());
72
73                if (p.needReference())
74                {
75                    fvc::makeRelative(phi, U);
76                    adjustPhi(phi, U, p);
77                    fvc::makeAbsolute(phi, U);
78                }
79
80                for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
81                {
82                   fvScalarMatrix pEqn
83                    (
84                        fvm::laplacian(rAU, p) == fvc::div(phi)
85                    );
86
87                   pEqn.setReference(pRefCell, pRefValue);
88
89                  if
90                    (
91                        ocorr == nOuterCorr-1
92                     && corr == nCorr-1
93                     && nonOrth == nNonOrthCorr)
94                    {
95
96                        pEqn.solve(mesh.solver(p.name() + "Final"));
97                    }
98                    else
99                    {
100                       pEqn.solve(mesh.solver(p.name()));
101                   }
102
103                   if (nonOrth == nNonOrthCorr)
104                   {
105                       phi -= pEqn.flux();
106                   }
107               }
```

```
108
109             #include "continuityErrs.H"
110
111             // Explicitly relax pressure for momentum corrector
112             if (ocorr != nOuterCorr-1)
113             {
114                 p.relax();
115             }
116
117             // Make the fluxes relative to the mesh motion
118             fvc::makeRelative(phi, U);
119
120             U -= rAU*fvc::grad(p);
121             U.correctBoundaryConditions();
122         }
123     //Acaba añadido
124             turbulence->correct();
125     }
126
127     runTime.write();
128
129     Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
130         << "  ClockTime = " << runTime.elapsedClockTime() << " s"
131         << nl << endl;
132 }
133
134 Info<< "End\n" << endl;
135
136 return 0;
130 }
// ************************************************************************* //
```

Código 6. Código numerado de *pimpleDyMFoam*

Como en el caso del *blockMeshDict*, y como todos los archivos de código de OpenFOAM, la primera parte del texto es una cabecera de color azul que contiene información sobre la versión del software y en algunas ocasiones una breve descripción sobre el archivo. En este caso la descripción -*description*- explica para qué tipo de flujo es adecuado el *solver* y las diferentes formas de tratar la turbulencia de las que se dispone -laminar, RAS, LES-.

El código própiamente dicho empieza en la línea numerada con un 1 después de la línea divisioria inferior de la cabecera. Las primera líneas -1 a 4- incluyen archivos que el *solver* necesita para poder ejecutarse de forma correcta. En otras palabras es como ampliar las funciones disponibles para el *solver*. Si, por ejemplo, no se incluyera la declaración de la línea 4, no se podría acceder a ninguna de las funciones que posee la clase *dynamicFvMesh*, que en este caso esta relacionada con el movimiento de malla.

En la línea 8 empieza el programa principal *-main-*, que en este caso está definido como un número entero, así que ese tipo de variable es lo que tendrá que devolver al usuario al final como se muestra en la línea 136. Las primera líneas dentro del programa principal también se dedican a incluir los diferentes tipos de clases necesarias para la ejecución.

El código hace que se muestre, al llegar a la línea 22, un mensage en la pantalla que indique que se empieza el bucle temporal en el que se realizarán las iteraciones numéricas. Es en la línea 24 cuando empieza realmente el bucle temporal del tipo *while*, lo que significa que el proceso se repetirá hasta que se cumpla una cierta condición. En este consiste en haber llegado al tiempo final de la simulación, indicado en el archivo *controlDict* dentro de la carpeta *system*. Óbviamente el bucle abortará en caso de que haya algún tipo de error así que no se llegará al tiempo final establecido en ese caso.

La primera función numérica del código se encuentra en la línea 30, donde se utiliza la función *makeAbsolute* incluída en el espacio *fvc*, un espacio dedicado al cálculo de derivadas explícitas. Esta declaración es un ejemplo de la claredad de código que se consigue en OpenFOAM gracias a su estructura en espacios y clases. En esta línea de código se convierte el flujo másico en absoluto, sin tener en cuenta el posible movimiento relativo de la malla.

El primer aspecto relacionado con la malla dinámica ocurre en la línea 38, momento en el que se actualiza la geometría de la malla. En el caso de que la malla se haya desplazado, el bucle de las líneas 40 a 43 lo detectará y realizará una corrección del flujo másico de cada una de las celdas *-phi-*.

Debido al movimiento de la malla, el flujo debe convertirse en relativo al movimiento de las celdas para conservar su validez mediante el código de la línea 46. El bucle *if* situado a continuación se ocupa de calcular de nuevo el número de Courant de la simulación para tener en cuenta el movimiento que ha habido en la malla. El número de Courant medio y máximo de la simulación ya se habían obtenido al principio del código, cuando no había movimiento de malla aún, en la declaración de la línea 27.

El bucle *pimple* que da parte del nombre a este *solver* ocupa de la línea 55 a la 125. Se trata de una mezcla de dos algoritmos llamados *piso* y *simple* que son en algunos aspectos bastante diferentes. La mayor discrepancia entre ambos consiste en el hecho de que *simple* utiliza factores de relajación y se utiliza para fluidos estacionarios turbulentos. De esta manera, en las ecuaciones del proceso simple no aparecen las derivadas temporales que si aparecen en piso. Sin embargo, una mezcla de los dos permite poder utilizar incrementos temporales mayores sin perder la convergencia de

resultados gracias a que realiza iteraciones dentro de cada paso de tiempo. La idea es que *piso* se encarga de realizar la parte relacionada con el flujo transitorio y *simple* ayuda a conseguir una mejor convergencia en cada uno de los incrementos temporales.

El hecho de utilizar *pimple* aumenta de forma significante el valor del incremento temporal posible para realizar la simulación. Este hecho se puede observar intentando resolver con *icoFoam* un tutorial que se trate con *pimpleFoam*. Si no se realiza ningúna modificación en el paso de tiempo lo más probable es que los errores vayan aumentando cada iteración hasta que se produzca un error -una divergencia habitualmente- y el *solver* deba abortar. Es precisamente por esta razón que las condiciones para finalizar el bucle en los procesos *simple* y *piso* son diferentes. Mientras el primero itera hasta la convergencia, el segundo itera hasta un cierto tiempo prescrito. Nótese que se está haciendo referencia a los procesos y no a los bucles temporales de los propios *solvers* que están regidos por *controlDict.*

Dentro del bucle *pimple* se llevan a cabo las siguientes acciones:

- Guardar la presión calculada en la iteración anterior porque es necesaria para aplicar la relajación -línea 59-.
- Definir la ecuación de Navier-Stokes para un flujo incompresible, con viscosidad constante y teniendo en cuenta la existencia de turbulencia -línea 62-. Se incluye a continuación el código correspondiente a esa línea con explicaciones para facilitar la comprensión.

  Se definen los términos de la ecuación de Navier-Stokes en los que aparece la velocidad *U*:

```
fvVectorMatrix UEqn

            (
                fvm::ddt(U)
              + fvm::div(phi, U)
              + turbulence->divDevReff(U)
            );
```

Código 7. Términos de Navier-Stokes incluyendo la velocidad.

  Se relajan los términos definidos en el código anterior, englobados en la variable *UEqn*:

```
if (ocorr != nOuterCorr-1)

            {
```

```
UEqn.relax();
                }
```

Código 8. Relajación de los términos del código 7.

Se resuelve la ecuación de Navier-Stokes. Recibe el nombre de predictor de cantidad de movimiento porque más adelante en el código se corregira su valor mediante otros resultados.

```
if (momentumPredictor)
            {
                solve(UEqn == -fvc::grad(p));
            }
```

Código 9. Se resuelve Navier-Stokes.

- Se obtiene un coeficiente discretizador y se calcula la velocidad y el flujo másico en las celdas -líneas 68, 70 y 71 respectivamente-. A continuación se corrige el flujo recientemente obtenido para tener en cuenta el efecto del movimiento de la malla -de la línea 75 a la 77-.
- Se define y se resuelve la ecuación de la presión -líneas 82 a 101- y se corrige el flujo a partir de este último cálculo en la línea 105.
- Una vez se ha calculado la presión, se le aplica un factor de relajación -línea 114- y se utiliza este valor para corregir la velocidad calculada mediante el predictor de cantidad de movimiento -líneas 118 a 121-.
- Después de efectuar un paso correctivo sobre la turbulencia en la línea 124, el código muestra varios mensajes por pantalla y vuelve a empezar el bucle pimple.

8.2.2 COMPARACIÓN DE *PIMPLEDYMFOAM* CON *PIMPLEFOAM*

Una vez se ha comprendido el funcionamiento básico de *pimpleDyMFoam* se debe comparar con *pimpleFoam* para localizar las diferencias desde el punto de vista de la malla dinámica. Algunos de los principales aspectos se pueden identificar fácilmente por el hecho de que aparece la propia palabra *mesh* -malla- en el código. Sin embargo, se debe de analizar el código -incluido a continuación- detalladamente para asegurarse de que no se omite ningún detalle.

```
0       #include "fvCFD.H"
1       #include "singlePhaseTransportModel.H"
2       #include "turbulenceModel.H"
3
4       // * * * * * * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * *
* //
5
6       int main(int argc, char *argv[])
7       {
8           #include "setRootCase.H"
9           #include "createTime.H"
10          #include "createMesh.H"
11          #include "createFields.H"
12          #include "initContinuityErrs.H"
13
14          Info<< "\nStarting time loop\n" << endl;
15
16          while (runTime.run())
17          {
18              #include "readTimeControls.H"
19              #include "readPIMPLEControls.H"
20              #include "CourantNo.H"
21              #include "setDeltaT.H"
22
23              runTime++;
24
25              Info<< "Time = " << runTime.timeName() << nl << endl;
26
27              // --- Pressure-velocity PIMPLE corrector loop
28              for (int oCorr=0; oCorr<nOuterCorr; oCorr++)
29              {
30                  if (nOuterCorr != 1)
31                  {
32                      p.storePrevIter();
33                  }
34
35                  #include "UEqn.H"
36
37                  // --- PISO loop
38                  for (int corr=0; corr<nCorr; corr++)
39                  {
40                      #include "pEqn.H"
41                  }
42
43                  turbulence->correct();
44              }
45
46              runTime.write();
47
48              Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
49                  << "  ClockTime = " << runTime.elapsedClockTime() << " s"
50                  << nl << endl;
51          }
```

```
52
53          Info<< "End\n" << endl;
54
55          return 0;
56      }
```

Código 10. Código numerado de *pimpleFoam*.

Esta vez se ha omitido la cabecera del código porque es exactamente la misma que en el caso de *pimpleDyMFoam* pero con un comentario diferente en la descripción. Además, el código es más corto porque toda la parte incluida entre las líneas 70 y 121 de *pimpleDyMFoam* se encuentra dentro del archivo incluido en la línea 40 llamado *pEqn.H*. Se deberá abrir dicho archivo para comparar esa parte con su equivalente en el otro *solver*.

Se observa que al principio del código, cuando se incluyen las filas auxiliares para el *solver*, no figura el *dynamicFvMesh.H*, por lo tanto, definitivamente esa es la clase que trata los casos de malla dinámica. Así que se deberá incluir esa línea en el *solver* a desarrollar.

En segundo lugar, también entre los archivos incluídos, se puede observar que se ha substituido *createMesh* por *createDynamicFvMesh*. Esto denota que desde la creación de la malla se debe utilizar otro tipo de funciones para tratar el movimiento de malla y que, por tanto, se trata del segundo cambio a introducir.

Los archivos de las líneas 18 y 19 se han substituído por uno llamado *readControls*. Esto a priori parece un cambio pero en realidad el código que se encuentra dentro de *readControls* incluye a los dos archivos eliminados lo que significa que no ha habido ninguna variación.

La declaración de la línea 30 de *pimpleDyMFoam* no aparece como era de esperar, ya que si no hay movimiento de malla el flujo siempre es absoluto. Lo mismo ocurre con el código que actualiza la geometría de la malla en la línea 38 y con el código entre las líneas 40 y 51 de *pimpleDyMFoam*, que en el caso de no haber malla dinámica no tiene sentido incluirlos.

La próxima parte que se debe comparar es el fichero *Ueqn.H* que en ambos casos se encuentra separado del código principal. Para *pimpleDyMFoam* ya se ha mostrado con anterioridad y en el caso de *pimpleFoam* es el siguiente:

```
tmp<fvVectorMatrix> UEqn

        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          + turbulence->divDevReff(U)
        );
        if (oCorr == nOuterCorr-1)
        {
            UEqn().relax(1);
        }
else
        {
            UEqn().relax();
        }

        volScalarField rUA = 1.0/UEqn().A();

        if (momentumPredictor)
        {
            if (oCorr == nOuterCorr-1)
        {
                solve(UEqn() == -fvc::grad(p), mesh.solver("UFinal"));
            }
            else
            {
         solve(UEqn() == -fvc::grad(p));
            }
        }
        else
        {
            U = rUA*(UEqn().H() - fvc::grad(p));
            U.correctBoundaryConditions();
        }
```

Código 11. Parte del código correspondiente al fichero *UEqn.H*.

Aunque se observan discrepancias entre los dos códigos, éstas no estan relacionadas con asuntos de malla dinámica por lo que se proseguirá el análisis en el otro archivo incluido: *Peqn.H*, que se muestra a continuación.

```
U = rUA*UEqn().H();

if (nCorr <= 1)

        {
            UEqn.clear();
        }

        phi = (fvc::interpolate(U) & mesh.Sf())
            + fvc::ddtPhiCorr(rUA, U, phi);
```

```
        adjustPhi(phi, U, p);
        // Non-orthogonal pressure corrector loop
        for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
        {
            // Pressure corrector
            fvScalarMatrix pEqn
            (

                fvm::laplacian(rUA, p) == fvc::div(phi)
            );
pEqn.setReference(pRefCell, pRefValue);
            if
            (
                oCorr == nOuterCorr-1
             && corr == nCorr-1
             && nonOrth == nNonOrthCorr
             )
            {
                pEqn.solve(mesh.solver("pFinal"));
            }
            else
            {
                pEqn.solve();
             }
             if (nonOrth == nNonOrthCorr)
            {
                phi -= pEqn.flux();
            }
        }

        #include "continuityErrs.H"

        // Explicitly relax pressure for momentum corrector except for last
        corrector
        if (oCorr != nOuterCorr-1)
        {
            p.relax();
        }

        U -= rUA*fvc::grad(p);
        U.correctBoundaryConditions();
```

Código 12. Código del archivo *pEqn.H*, dentro de *pimpleFoam*.

No se observan más diferencias entre *pimpleDyMFoam* y *pimpleFoam* en referencia al tratamiento del desplazamiento de la malla. Así que, a modo de resumen, las modificaciones que se deberán realizar a icoFoam serán:

- Incluir *dynamicFvMesh.H*

- Substituir *createMesh.H* por *createDynamicFvMesh.H*
- Incluir la actualización de la geometría de malla y las consiguientes correcciones de flujo y de número de Courant, teniendo en cuenta que se debe de realizar la operación de convertir los flujos de relativos a absolutos y viceversa.

Una vez identificada toda la parte de código necesaria para tratar la malla dinámica, el siguiente paso es comprobar que los archivos incluidos que se encuentran dentro de la carpeta del *solver* están correctamente definidos. Los *solvers* se encuentran dentro de la carpeta *applications/solvers*, donde están organizados en función de las características del flujo que tratan.

En el caso de *pimpleDyMFoam* encontramos los archivos *readControls.H*, *correctPhi.H*, *createFields.H* y *UEqn.H*. Por otro lado, *IcoFoam* solamente contiene el archivo *createFields.H*, de modo que se deberán añadir los archivos *correctPhi.H*, relacionado con la corrección del flujo másico debido al movimiento de la malla, y *readControls.H*, que se basa en parte del código de *correctPhi.H*. El archivo *UEqn.H* no hace falta incluirlo porque es el que se ocupa de definir la ecuación de cantidad de movimiento y en el caso de *icoFoam* ya está incluida en el código principal.

El archivo *correctPhi.H* se puede copiar tal cual, pero el *readControls.H* necesita varias modificaciones. Lo que realmente se debe realizar es incluir el código de *readControls.H* sin tener en cuenta las inclusiones de archivos, es decir, añadir lo siguiente -extraído de la carpeta de *pimpleDyMFoam*-.

```
bool correctPhi = false;

    if (pimple.found("correctPhi"))
    {
        correctPhi = Switch(pimple.lookup("correctPhi"));
    }

    bool checkMeshCourantNo = false;
    if (pimple.found("checkMeshCourantNo"))
    {
        checkMeshCourantNo = Switch(pimple.lookup("checkMeshCourantNo"));
    }
```

Código 13. Aspecto del código de *readControls.H* que se debe incorporar a *icoDyMFoam*.

Sin embargo, en el caso de *icoDyMFoam* se debe substituir la palabra *pimple* por *piso* todas las veces que sale para adecuar esta parte de código al proceso *piso*, que será el seguido por el nuevo *solver*. Además para que el código anterior funcione se debe incluir el archivo *readPISOControls.H*, en lugar de *readPIMPLEControls.H*, por tratarse de un

proceso *piso*.

El archivo *readTimeControls.H*, que se incluye en el código de *readControls.H* de *pimpleDyMFoam*, no es necesario para *icoFoam* porque éste son experimenta variaciones en el incremento temporal. Por el contrario, *pimpleDyMFoam* lo modifica para mantener constante el máximo número de Courant de la simulación. Concretamente, lo reduce de forma inmediata y lo va aumentando progresivamente para evitar oscilaciones inestables.

8.2.3 MODIFICACIÓN DE ICOFOAM

Con los cambios mencionados en el apartado anterior, el nuevo *solver* llamado *icoDyMFoam* tiene el siguiente código:

```
    #include "fvCFD.H"
    #include "dynamicFvMesh.H"

    // * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
     int main(int argc, char *argv[])
    {
        #include "setRootCase.H"
        #include "createTime.H"
       #include "createDynamicFvMesh.H"
        #include "initContinuityErrs.H"
        #include "createFields.H"

        // * * *  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
Info<< "\nStarting time loop with fascinating new icoDyMFoam\n" << endl;
  while (runTime.loop())
                      (
                          fvm::laplacian(rAU, p) == fvc::div(phi)
                      );

                      pEqn.setReference(pRefCell, pRefValue);
                     pEqn.solve();
            if (nonOrth == nNonOrthCorr)
                  {
                            phi -= pEqn.flux();
                  }
           }

      #include "continuityErrs.H"

      // Make the fluxes relative to the mesh motion
      fvc::makeRelative(phi, U);
```

```
U -= rAU*fvc::grad(p);
U.correctBoundaryConditions();
}
{
   Info<< "Time = " << runTime.timeName() << nl << endl;
       #include "readControls.H"
       #include "CourantNo.H"

       // Make the fluxes absolute
       fvc::makeAbsolute(phi, U);

      // Do any mesh changes
       mesh.update();

       if (mesh.changing() && correctPhi)
       {
      #include "correctPhi.H"

       }

       // Make the fluxes relative to the mesh motion
       fvc::makeRelative(phi, U);

       if (mesh.changing() && checkMeshCourantNo)
       {
        #include "meshCourantNo.H"
       }

       fvVectorMatrix UEqn
       (
           fvm::ddt(U)
         + fvm::div(phi, U)
         - fvm::laplacian(nu, U)
       );

       if (momentumPredictor)
      {
         solve(UEqn == -fvc::grad(p));
      }
   // --- PISO loop

       for (int corr=0; corr<nCorr; corr++)
           {
               rAU = 1.0/UEqn.A();

               U = rAU*UEqn.H();
               phi = (fvc::interpolate(U) & mesh.Sf())
                + fvc::ddtPhiCorr(rAU, U, phi);

               adjustPhi(phi, U, p);
       for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
       {
                   fvScalarMatrix pEqn
```

```
                                runTime.write();

            Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
             << "  ClockTime = " << runTime.elapsedClockTime() << " s"
                    << nl << endl;
        }

        Info<< "End\n" << endl;

        return 0;
    }

    //************************************************************** //
```

Código 14. Aspecto de *icoDyMFoam*.

Que si se compara con el original *icoFoam* -mostrado abajo- se observa que solo se han añadido los cambios mencionados en el apartado 8.2.2.

```
#include "fvCFD.H"

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
int main(int argc, char *argv[])
    {
        #include "setRootCase.H"
        #include "createTime.H"
        #include "createMesh.H"

        #include "initContinuityErrs.H"
        #include "createFields.H"

     // * * *  * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

        Info<< "\nStarting time loop\n" << endl;

        while (runTime.loop())
        {

            Info<< "Time = " << runTime.timeName() << nl << endl;
            #include "readPISOControls.H"
            #include "CourantNo.H"


            fvVectorMatrix UEqn
            (
                fvm::ddt(U)
              + fvm::div(phi, U)
              - fvm::laplacian(nu, U)
            );
      if (momentumPredictor)
          {
```

```
            solve(UEqn == -fvc::grad(p));
        }
// --- PISO loop
        for (int corr=0; corr<nCorr; corr++)
        {
            volScalarField rUA = 1.0/UEqn.A();
            U = rUA*UEqn.H();
            phi = (fvc::interpolate(U) & mesh.Sf())
            + fvc::ddtPhiCorr(rUA, U, phi);
            adjustPhi(phi, U, p);

            for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
            {
                fvScalarMatrix pEqn
                (
                    fvm::laplacian(rUA, p) == fvc::div(phi)
                );

                pEqn.setReference(pRefCell, pRefValue);
                pEqn.solve();
            if (nonOrth == nNonOrthCorr)
            {
                phi -= pEqn.flux();
            }
            }

            #include "continuityErrs.H"

            U -= rUA*fvc::grad(p);
            U.correctBoundaryConditions();
        }

        runTime.write();
        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;
    return 0;
}
```

Código 15. Código original de *icoFoam*.

Hasta el momento, ya está el código principal del nuevo solver preparado. Como se ha indicado anteriormente, el siguiente paso es arreglar los archivos incluidos dentro de su carpeta. Entre ellos, *CorrectPhi.H* se copia directamente, y *readControls.H* -incluido a continuación- se añade con los cambios mencionados anteriormente.

```
#include "readPISOControls.H"

            bool correctPhi = false;
            if (piso.found("correctPhi"))
            {
                correctPhi = Switch(piso.lookup("correctPhi"));
            }

            bool checkMeshCourantNo = false;
            if (piso.found("checkMeshCourantNo"))
            {
                checkMeshCourantNo = Switch(piso.lookup("checkMeshCourantNo"));
            }
```

Código 16. *ReadControls.H*, tal y como se debe de incluir en *icoDyMFoam*.

Cabe mencionar que hay una diferencia entre *icoDyMFoam* y *icoFoam* que hace que se tenga que modificar el archivo *createFields.H* a incluir en la carpeta de *icoDyMFoam.* En el *solver* original la variable *rUA -rAU* en *icoDyMFoam*- no se define hasta llegar al bucle *piso*. Por otro lado, en *icoDyMFoam* esta variable aparece antes, concretamente en el archivo incluido *correctPhi.H*. Por esta razón se necesita que dicha variable se defina desde el principio, como es el caso de la velocidad y la presión en este tipo de *solvers*, y eso se realiza dentro de *createFields.H*, que es el archivo responsable de crear las variables iniciales.

```
Info<< "Reading field rAU\n" << endl;

            volScalarField rAU
            (
                IOobject
                (
                    "rAU",
                    runTime.timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::AUTO_WRITE
                ),
                mesh,
                runTime.deltaT(),
                zeroGradientFvPatchScalarField::typeName
            );
```

Código 17. Declaración del campo *rAU* dentro de *createFields.H*.

Una vez la carpeta del *solver* ya contiene todos los archivos necesarios y el código está acabado, el siguiente paso es la compilación.

Básicamente, mediante la compilación lo que se realiza es una comprobación de que

tanto el código como los archivos incluidos son correctos. Si la compilación finaliza con éxito se puede decir que el solver ya existe y está preparado para ser ejecutado. Si hay algún error aparecerá en pantalla una descripción que indica de qué se trata y en qué archivo se encuentra de modo que su subsanación no será excesivamente complicada. Para llevar a cabo la compilación se debe acceder con el terminal a la carpeta que contiene el *solver* y ejecutar *wmake*.

8.2.4 VALIDACIÓN DE ICODYMFOAM

Una vez la compilación se ha realizado con éxito el nuevo *solver* ya está preparado para ser utilizado. Para comprobar que funciona correctamente se escogerá un tutorial de *pimpleDyMFoam* y se resolverá con *icoDyMFoam*. El objetivo es únicamente poner en marcha el *solver* por primera vez y asegurarse de que los resultados obtenidos tienen sentido. Más adelante, una vez ya se haya validado el funcionamiento de *icoDyMFoam*, se procederá a profundizar en aspectos sobre el movimiento de la malla.
El tutorial elegido es *movingCone*. Un dominio axisimétrico en el que se desplaza una pared trapezoidal que fuerza al fluido delante de ella a pasar por un estrecho margen en la parte superior. Lo primero que se debe hacer es copiar el caso entero, que se llamará *icomovingCone*, y luego intentar directamente ejecutar el nuevo *solver*. De esta manera, OpenFOAM nos indicará qué tipo de cambios hace falta realizar para que todos los archivos de apoyo en las carpetas *constant* y *system* sean los adecuados. Ejecutando *icoDyMFoam* en *icomovingcone* el resultado es el siguiente:

```
--> FOAM FATAL IO ERROR:
keyword PISO is undefined in dictionary
"/home/.../icomovingCone/system/fvSolution"
```

Como se ha mencionado, OpenFOAM describe el error diciendo que en el archivo *fvSolution* no se encuentra la palabra *piso*. Si se abre el archivo se observa que la palabra que aparece es *pimple* porque ese es el proceso utilizado por *pimpleDyMFoam*. Sin embargo, ahora el proceso del solver es *piso* y esta es la palabra que debe aparecer. Así que se debe cambiar la palabra y lo que va después de ella se obtiene del *fvSolution* de un caso cuyo solver utilize el proceso *piso*. Con la modificación, el caso funciona y se realizan las primeras iteraciones.

Es de especial importancia comprobar que el movimiento de la malla corresponde exactamente con el del tutorial *movingCone*. Si los resultados coinciden quedará demostrada la correcta manipulación del movimiento de la malla por parte del *solver*

creado. Para observar los resultados se ejecuta *paraFoam*, comando que nos abre una interfaz gráfica que permite ver los resultados para cada carpeta temporal creada, tanto de la malla como de los campos calculados.

En la figura 19 se observa la deformación de la malla en varios instantes del tutorial original y del caso idéntico simulado con el nuevo *solver*. Se puede observar que la malla ha sido tratada exactamente igual en ambos casos, por lo tanto, en este aspecto se considera que icoDyMFoam funciona correctamente.

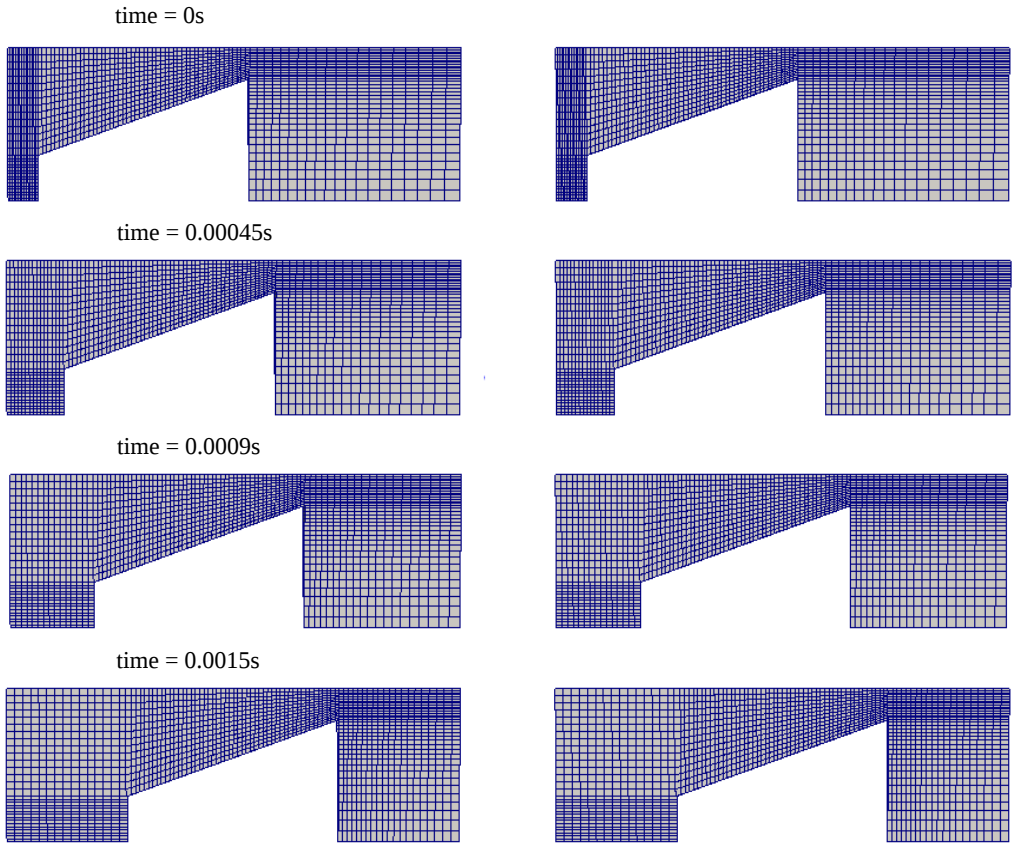


Figura 19. Movimiento de malla en el caso *movingCone* para varios instantes temporales de la simulación. La columna de la izquierda muestra el obtenido con *pimpleDyMFoam*; la de la derecha el obtenido con *icoDyMFoam*.

El siguiente paso para validar la nueva aplicación es comprobar que los resultados de los campos calculados son correctos. Para llevar a cabo esta parte se resuelve un mismo caso con dos *solvers* distintos: *icoFoam* y *icoDyMFoam.* Como el objetivo desde el principio era crear una aplicación idéntica a *icoFoam* pero con la capacidad añadida de tratar el movimiento de malla, los resultados deben de ser idénticos. Obviamente se debe de aplicar el análisis a un caso en el que no haya movimiento de malla porque *icoFoam* no

es capaz de resolverlo. Además, la efectividad del nuevo solver en relación con la deformación de la malla ya ha quedado reflejada anteriormente.

El caso elegido es *obstacleCavity.* Se trata de una copia del original *cavity* con un obstáculo cuadrado en el centro como única modificación geométrica. Es un dominio cuadrado en el que la pared superior se desplaza horizontalmente a una velocidad de 5m/s mientras que en el resto de paredes se impone velocidad nula en todas sus componentes. Se ha decidido utilizar este caso porque ha sido el primero en ser utilizado para obtener resultados con malla dinámica. *Cavity* es un conocido tutorial muy utilizado para mostrar los principales aspectos de OpenFOAM y es normalmente el primero en ser manipulado por principiantes.

Al realizar la simulación con *icoDyMFoam* se debe de imponer que la velocidad de todas las fronteras del dominio es nula. En este caso el *solver* también calcula el movimiento de los puntos pero su valor es nulo para todas las iteraciones. Así, la aplicación actuará como si fuera *icoFoam* y permitirá realizar una buena comparación.

En la imagen inferior -figura 20- se puede observar que los resultados obtenidos por cada uno de los *solvers* son idénticos. Se muestra el valor en magnitud de la velocidad representado por una escala de colores y su dirección representada mediante las flechas. En ambos casos las magnitudes y direcciones coinciden. Además los valores límite de la escala de velocidades, dato que calcula automáticamente OpenFOAM, son idénticamente iguales para ambos casos.

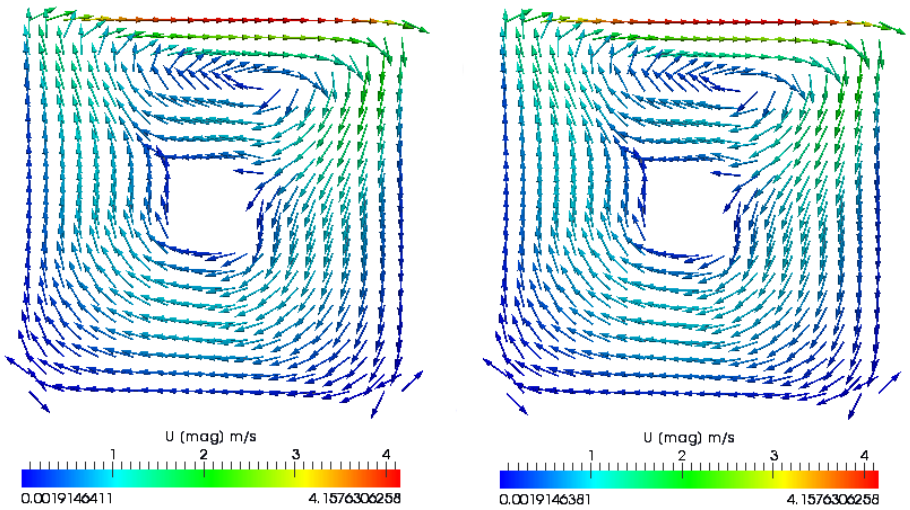


Figura 20. Resultados del campo de velocidades para el caso *cavity* con un obstáculo cuadrado en el centro. A la izquierda se muestran los obtenidos con *icoFoam*; a la derecha los de *icoDyMFoam*.

Las dos comprobaciones realizadas demuestran que *icoDyMFoam* es equivalente a *icoFoam* y que trata el movimiento de malla de forma adecuada. Por lo tanto, se considera que *icoDyMFoam* es válido y se procederá a emplear para las consiguientes simulaciones relacionadas con el movimiento de malla.

8.3 MOVIMIENTO LINEAL DEL OBSTÁCULO

Una vez se ha añadido con éxito el obstáculo en el caso cavity el siguiente objetivo es conseguir que se mueva de alguna manera mediante *icoDyMFoam*. En esta ocasión, como en todas las modifiaciones realizadas en el estudio, se buscará un tutorial que incluya aspectos lo más similares posibles al tipo de movimiento que se pretende conseguir: rotación. La aplicación con malla dinámica que se considera más adecuada es *movingCone*, un caso que incluye la translación horizontal de una pared. Se ha elegido un caso con desplazamiento en lugar de giro porque no hay ningún ejemplo que incluya rotación de objetos en la distribución 1.7.1 de OpenFOAM.

El proceso consistirá en hacer una copia del caso *cavity* con obstáculo -llamado *obstacleDyMCavity*- y ir modificándolo paso a paso hasta conseguir el movimiento del cuadrado central. Se comparará con *movingCone* para poder decidir qué aspectos deben incorporarse y de qué manera. Posteriormente, el esfuerzo se centrará en intentar cambiar la translación por una rotación.

Se empieza el análisis en la carpeta temporal incial -0- de ambos casos, donde se observa que en *movingCone* existen dos variables añadidas a parte de la presión y la velocidad. Son las encargadas de aplicar velocidad exclusivamente horizontal -dirección x- a los puntos y a las celdas del dominio y se denominan respectivamente *pointMotionUx* y *cellMotionUx*. Hay que copiar los dos archivos al caso nuevo -*obstacleDyMCavity*- y adaptarlos a su geometría, es decir, modificar el nombre de las fronteras y sus condiciones de contorno. Concretamente, es necesario que se modifique la definición de las fronteras de *obstacleDyMCavity* porque ahora, la pared superior será tratada como una fija en lugar de una móvil. Solo las paredes que corresponden al obstáculo van a desplazarse, el resto se mantendrán fijas. Si no se tiene en cuenta esto, los puntos que se moverán no serán los del cuadrado, sino los de la superficie superior.

Se observa que el caso original es axisimétrico, porque tiene una frontera definida con la palabra *wedge.* Como la geometría de *obstacleDyMCavity* se adapta mejor a un caso 2D, se substituirá *wedge* por *empty*, que es la palabra para definir las fronteras sobre las que

se extiende la profundidad en casos bidimensionales.

A continuación se compara el contenido de las carpetas *constant* de ambos casos. Como era de esperar, en *movingCone* aparece el archivo que gobierna el movimiento de la malla: *dynamicMeshDict*. Por ahora, el objetivo es simplemente imitar el movimiento horizontal así que se copiará de un caso al otro sin modificaciones. Por otro lado, *movingCone* incluye un archivo que trata la modelización de la turbulencia, pero se puede obviar porque no se desea incluir turbulencia en la posterior simulación con *icoDyMFoam*. Finalmente, cabe decir que en ambos casos se encuentra el archivo *transportProperties*, dedicado a definir las propiedades físicas del flujo. El número de propiedades que se definen depende del tratamiento que se vaya a realizar por parte del *solver*. Para *icoDyMFoam* la única propiedad que importa es la viscosidad cinemática, y ésta ya está incluida así que la carpeta *constant* también está preparada.

La última carpeta que queda por comparar es *system*. Dentro de ella, el archivo *controlDict* no representa ningún interés ya que solo afecta a parámetros como el incremento temporal, así que se centra la atención en los dos restantes: *fvSchemes* y *fvSolution*. Encontrar cuáles son las diferencias y aplicarlas debidamente a *obstacleDyMCavity* es tedioso y puede llevar bastante tiempo. Por eso, se prepararán estos dos archivos a través de los errores que aparecen en el momento de ejecutar la simulación. Normalmente son pequeñas modificaciones y el propio OpenFOAM indica en que parte del documento se deben de incorporar.

Antes de ejecutar el nuevo caso es muy importante observar el intérvalo de tiempo en el que se llevan a cabo ambas simulaciones. Puede que en uno solo sea de pocos milisegundos y en el otro de segundos. Esto puede afectar más severamente en los casos con malla dinámica porque puede ocurrir que una superficie se acerque demasiado a otra. Para evitar errores de este tipo se utilizará el mismo intérvalo e incremento temporal que utiliza el *movingCone* original, así como la misma velocidad de translación del obstáculo: 1m/s.

Si tratamos de ejecutar *icoDyMFoam*, empezarán a ser mencionadas todas las modificaciones que faltan para que el caso funcione correctamente. Las alteraciones a realizar son esquemas y *solvers* que se deben de añadir a *fvSchemes* o a *fvSolution* respectivamente. Todos ellos se copian directamente del caso *movingCone*. A continuación se muestran todos ellos:

Solvers:

- *cellMotionUx*

- *pFinal*

Esquemas:

- *laplacian(diffusivity, cellMotionU)*
- *laplacian(rAU,p)*

Con todos los cambios realizados ya se puede ejecutar sin errores la simulación. Como se ve en la figura 21 -*a* y *b*- los resultados son un poco pobres. El movimiento del cuadrado y la deformación de la malla son pequeños y no ilustran con detalle el comportamiento de los puntos. La velocidad del cuadrado se podría aumentar para causar una mayor deformación y así observar mejor el desplazamiento de la malla. Además, parece conveniente realizar una modificación en las condiciones de frontera para permitir que los puntos en las paredes superior e inferior se deslizen. Para eso hay que juntarlas en una nueva frontera que en las variables de movimientos de puntos será declarada como *slip* -deslizante-. Las variables de velocidad y presión se tratan de la misma forma que antes.

Al aplicar los cambios seleccionados, habiendo impuesto una velocidad de 3m/s al obstáculo, y ejecutar el *solver* se observa un movimiento de malla mucho más adecuado que el resultado anterior. Además, la mayor velocidad permite observar una deformación más considerable que permite evaluar mejor el comportamiento obtenido. En las partes *c* y *d* de la figura 21 se ilustra el cambio de los resultados de la simulación debido a las modificaciones mencionadas.

La distribución de velocidades es correcta y adopta en ambos casos unos valores que se pueden considerar realistas. El campo de presiones no se comprueba porque la velocidad depende de él y, si uno de los dos es válido, el otro también. El principal inconveniente se observa en el caso *b* de la figura 21, donde se intuye que si el movimiento del cuadrado fuera más amplio podrían aparecer errores por la excesiva deformación que aparecería. El hecho de permitir que los puntos de las superficies superior e inferior deslicen mejora significantemente la calidad de la malla como queda demostrado en el caso *d* de la misma imagen. Otra alternativa para mejorar el comportamiento de la malla es el cambio de la difusividad en la ecuación de Laplace que lo gobierna. Por ejemplo, imponiendo un esquema inversamente proporcional se consigue que los puntos más cercanos al obstáculo tengan unas uniones muy rígidas y los lejanos muy flexibles. La figura 22 muestra los exitosos resultados de esta distribución de difusividad incluso sin permitir el deslizamiento de puntos.

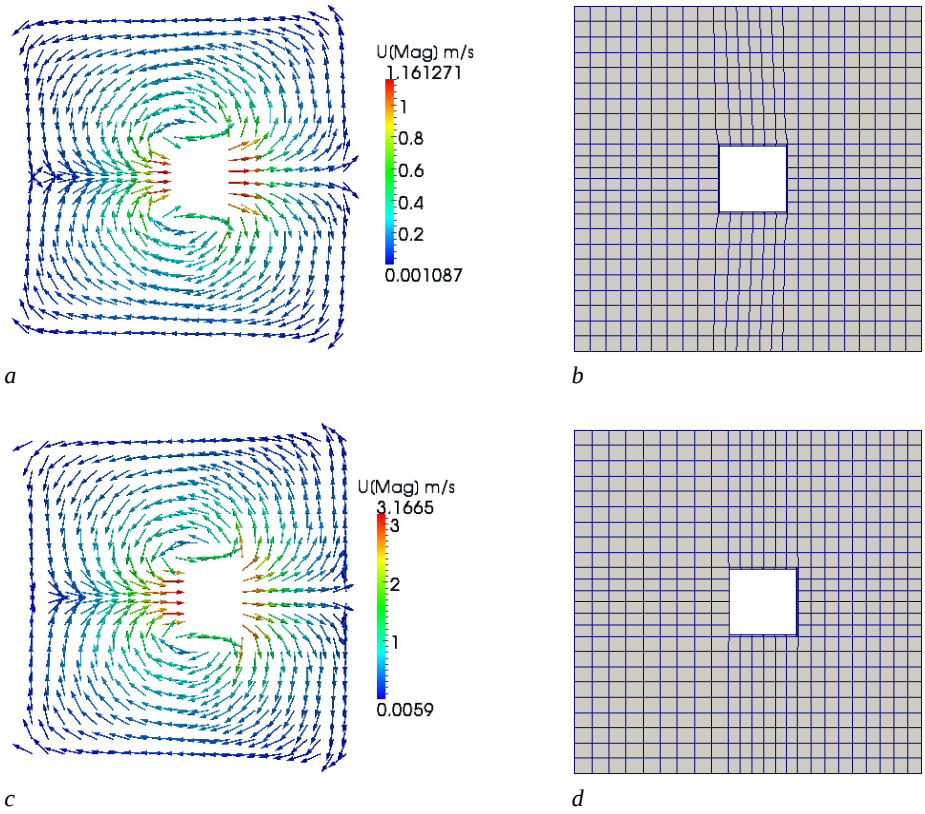


Figura 21. Resultados del movimiento de malla y el campo de velocidades del caso *cavity* con un obstáculo que se traslada horizontalmente. *a* y *b*: en las condiciones del caso original -*movingCone*-. *c* y *d*: con mayor velocidad de obstáculo -3m/s- y permitiendo que los puntos de las superficies superior e inferior deslizen.

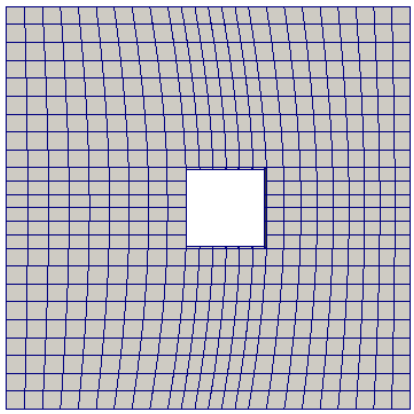


Figura 22. Movimiento de la malla en el caso *cavity* con obstáculo en translación horizontal. Esquema de difusividad inverso a la distancia y puntos de la frontera sin deslizar. Velocidad del cuadrado de 3m/s.

8.4 SOLVER DE DEFORMACIÓN DE MALLA POR COMPONENTES

Una vez se ha conseguido desplazar el obstáculo satisfactoriamente, el siguiente paso es lograr que éste rote. Lo que interesa es entender cómo funciona el solver de movimiento de malla utilizado y cuál es su relación con los archivos en la carpeta *0* de los que se alimenta. En este caso, como muestra el *dynamicMeshDict* de la carpeta *constant* – ver código 18-, el solver utilizado es *velocityComponentLaplacian*, que trabaja con variables escalares correspondientes a alguna de las direcciones de desplazamiento: x, y o z. No ofrece la posibilidad de que alguna de las componentes corresponda a un giro sobre cierto eje por lo que se deberá de buscar un *solver* más adecuado.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.7.1                                 |
|  \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|   \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      binary;
    class       dictionary;
    location    "constant";
    object      dynamicMeshDict;
}
// * * * * * *  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dynamicFvMesh   dynamicMotionSolverFvMesh;
motionSolverLibs ( "libfvMotionSolvers.so" );
solver          velocityComponentLaplacian x;
diffusivity     inverseDistance (movingWall);//directional ( 0.1 200 0 );
// ************************************************************************* //
```

Código 18. Ejemplo de *dynamicMeshDict*.

De las seis opciones de *solvers* de tratamiento de malla disponibles en la distribución 1.7.1 de OpenFOAM la más apropiada para imponer de la manera más fácil una rotación es *velocityLaplacian*. A diferencia del utilizado para el desplazamiento horizontal -*velocityComponentLaplacian*-, trabaja con variables de tipo vector. Esto conlleva que la variable *pointMotionUx* sea un vector en el que se pueden definir movimientos de malla con tres componentes.

El primer paso es cambiar el nombre del *solver* en *dynamicMeshDict* a *velocityLaplacian*. Para encontrar los cambios que se deben de realizar se intenta ejecutar la simulación con esta única modificación. Entonces, aparece un error que indica que no

se ha encontrado el archivo *pointMotionU*. Esto es porque en el existente hay una *x* añadida al final así que se procede a eliminiar la letra y a volver a intentarlo. El siguiente mensaje comunica que el *solver* espera recibir un *pointMotionU* que sea vector y está recibiento un escalar. Para remediar esta situación se debe de acceder a la cabecera de dicho archivo y reemplazar *pointScalarField* por *pointVectorField*, quedando la parte superior del documento como se muestra en el código 19.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       pointVectorField;
    object      pointMotionU;
}
```

Código 19. Cabecera del archivo *pointMotionU* para el *solver* vectorial.

No hace falta volver a intentar ejecutar la simulación para observar la necesidad de definir los valores dentro de *pointMotionU* mediante tres componentes, no una como en el caso de las translaciones. Basta con añadir dos componentes más a la preexistente y englobar los tres números, que están separados por un espacio, dentro de un paréntesis.

OpenFOAM continúa lanzando mensajes de error, el siguiente pide que se elimine la letra *x* en la palabra *cellMotionUx* dentro del archivo *fvSolution* porque ya se ha abandonado la situación en la que solo se estaba definiendo la componente horizontal del desplazamiento. Es el último error que aparece antes de que la simulación se pueda llevar a cabo sin problemas. El resultado es idéntico al obtenido antes de cambiar de *solver*, pero ahora el movimiento de la malla lo trata otro *solver* que es capaz de tratar translación en las tres componentes. Para comprobarlo, se añade un desplazamiento vertical de la misma magnitud que el horizontal y se muestra el resultado en la figura 23.

Se observa que para el nuevo solver de movimiento de malla no se necesita el antiguo archivo cellMotionUx de la carpeta temporal inicial. Ningún mensaje de error ha hecho referencia a dicho archivo, por lo tanto, se puede eliminar sin tener repercusión alguna en el funcionamiento de la simulación o en los resultados. De hecho, es posible que en el caso tratado con velocityComponentLaplacian este archivo tampoco tuviera ninguna función, pero esa comprobación carece de interés y no se va a realizar.

Óbviamente, la solución obtenida no sería la más adecuada si el movimiento del cuadrado fuera más amplio. Los puntos se encogerían o expandirían demasiado según su

localización en el dominio y podrían aparecer errores de discretizaciñon debida a la mala distribución de los puntos La mejor alternativa en este caso reside en permitir el deslizamiento de todos los puntos de la frontera exterior. De esta manera las celdas sufren una torsión menor, aguantando básicamente la compresión o expansión debido al movimiento del cuadrado. Si se compara la situación de la malla en este caso -figura 24- con el anterior -sin deslizamiento de puntos-, se observa una mejora substancial gracias a que los puntos se han deformado en menor medida.

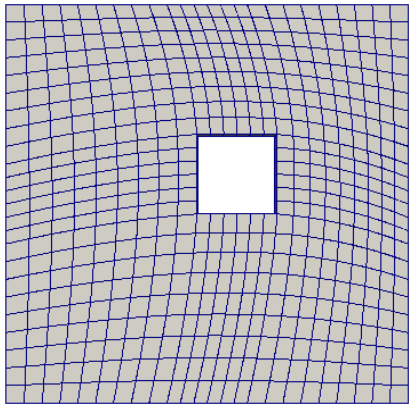


Figura 23. Movimiento de malla del caso *cavity* con un obstáculo en translación horizontal y vertical. La velocidad del cuadrado se han aumentado a 5m/s.

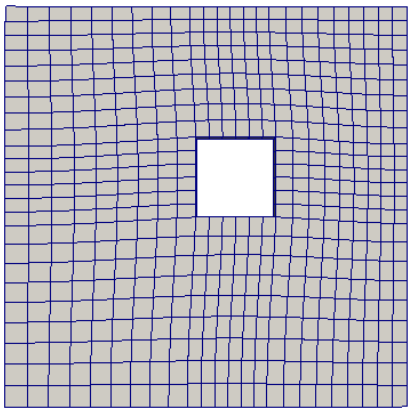


Figura 24. Movimiento de malla en el caso *cavity* con un obstáculo en translación horizontal y vertical. La velocidad del cuadrado es de 5m/s y los puntos de la frontera deslizan.

Cuanto mayor sea el desplazamiento del cuadrado menor será la capacidad de la malla de soportarlo. En realidad se puede obtener la posición a partir de la que la malla empieza a

sufrir graves daños. Existe una aplicación llamada *moveDynamicMesh* que simula únicamente la parte relacionada com el movimiento de los puntos. Además, efectúa comprobaciones y muestra ciertos parámetros de calidad de malla en cada incremento temporal. Si aumentamos la velocidad del cuadrado progresivamente manteniendo el tiempo total de la simulación, llegará un punto en el que el cuadrado estará tan cerca de la frontera que la malla estará exageradamente distorsionada.

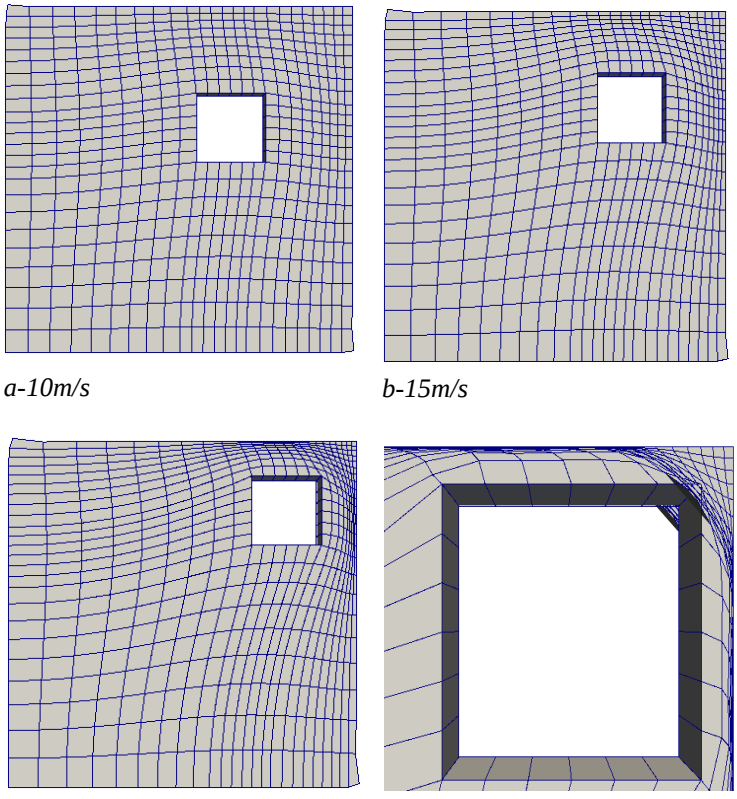


Figura 25. Movimiento de malla en el caso *cavity* con obstáculo en translación horizontal y vertical. La velocidad de las componentes horizontal y vertical del movimiento se indican al pie de cada imagen. El caso *d* es una vista más ampliada.

A medida que la posición final del cuadrado avanza hacia la esquina superior derecha, peor es el comportamiento de la malla según se observa en la figura 25. Cabe mencionar que incluso en el caso más lento, las esquinas de la frontera se ven deformadas porque sus puntos correspondientes se ven obligados a deslizar por la ley de difusividad impuesta. Esto no tiene gran efecto si lo que interesa es lo que ocurre en el alrededor más próximo al obstáculo pero es sin duda un error que el dominio se vea deformado de una manera no deseada.

El único caso de los de la figura 25 en el que *moveDynamicMesh* informa sobre errores en la malla es en el más rápido: 25m/s. El problema ya aparece antes de llegar a la situación ilustrada y consiste en la invasión del obstáculo por parte de la malla, es decir, una zona externa al dominio. En estos casos es muy complicado que el *solver* encargado de solucionar el flujo, de forma completamente independiente del movimiento de la malla, pueda obtener valores adecuados y lo más habitual es los errores detengan la simulación.

En los casos tratados anteriormente en los que el movimiento del cuadrado tiene componente vertical la difusividad estaba establecida en el esquema de inversidad proporcional. Para demostrar la robustez de dicha distribución de difusividad, se incluyen a continuación los resultados -figura 26- de la deformación de malla para el mismo caso tratados mediante un esquema direccional.

En este caso la malla produce errores muy temprano: para una velocidad de las componentes horizontal y vertical de 10m/s, situación cómodamente soportada previamente. Esto hace que se sitúe la variación de difusividad de manera inversamente proporcional a la distancia a una superficie como la opción preferida para el resto del estudio. La variacion de la difusividad de forma direccional no es adecuada para tratar este tipo de deformación de malla, sino para aquellos en los que solo hay movimiento en una sola dirección como era el caso de movingCone.

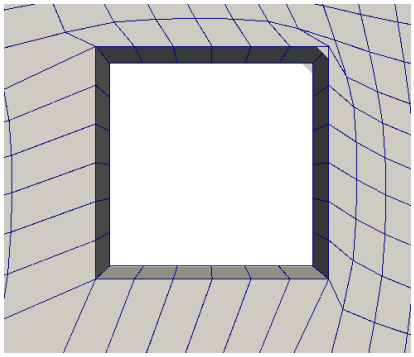


Figura 26. Movimiento de malla en el caso *cavity* con obstáculo en translación horizontal y vertical. La velocidad del cuadrado de 10m/s y el esquema de difusividad utilizado es el direccional.

Llegado este punto, interesa conocer cuáles son todas las condiciones de frontera posibles para la variable *pointMotionU* y escoger la que represente un movimiento rotativo. Para que aparezca un listado de todas las opciones posibles se debe de ejecutar la simulación con un error en el nombre de alguna condición de frontera dentro del

archivo *pointMotionU*. OpenFOAM lo detectará y lanzará un mensaje en el que se mostrarán todas las alternativas posibles -22-. De todas ellas, las únicas que imponen giro son *angularOscillatingDisplacement* y *angularOscillatingVelocity*. Pero como su nombre indica, se trata de oscilaciones respecto a una cierta posición de equilibrio, no de velocidades constantes. De modo que se debe crear una nueva condición de frontera para conseguir el movimiento deseado.

8.5 IMPLEMENTACIÓN DE UNA NUEVA CONDICIÓN DE FRONTERA

Las condiciones de frontera permiten definir el comportamiento de determinadas superfícies de la geometría del problema. Pueden aportar tanto propiedades físicas como movimientos y es necesario definirlas antes de empezar la simulación. Además, se debe asegurar que la configuración elegida es coherente o de lo contrario puede que sea imposible obtener resultados válidos, por ejemplo: que sea imposible satisfacer la continuidad.
OpenFOAM ofrece de antemano determinadas condiciones de frontera comunes para que los usuarios no tengan que implementarlas o, en caso que sea necesario crear nuevas, que haya una referencia que evite empezar de cero. Dichas condiciones se deben de aplicar a todas las superficies de la geometría y para cada uno de los campos que se vayan a calcular, por ejemplo, en el caso de que solo se vayan a obtener resultados para la presión y la velocidad se deben fijar condiciones de frontera para cada una de las dos propiedades mencionadas. El hecho de que los campos pueden ser de distinta naturaleza -vectores o escalares principalmente- provoca que las condiciones de frontera sean diferentes para cada caso, de modo que existen varios grupos de condiciones de frontera en función del tipo de variable que se calculará.

Las condiciones de frontera se definen en la primera carpeta temporal de la simulación. Allí debe de haber un archivo por cada campo a calcular donde se determinará el tipo de condición de frontera que se desea imponer sobre cada una de las superfícies definidas.

En el caso en el que se centra este estudio las condiciones de frontera para la velocidad y la presión se encuentran fácilmente entre las que ofrece OpenFOAM. Sin embargo, para el movimiento de la malla, no existe una condición que represente una rotación pura del cuerpo, de modo que se debe crear y compilar una nueva condición que lo cumpla. El movimiento más semejante se encuentra en una condición de frontera llamada *angularOscillatingVelocity*, que representa un movimiento rotativo oscilatorio. La parte más interesante de su código es la que define la oscilación que se impondrá sobre la geometría. Se muestra a continuación:

```
scalar angle = angle0_ + amplitude_*sin(omega_*t.value());
```

Código 20. Parte de la condición de frontera existente que se pretende modificar.

Por lo tanto, eliminando de la ecuación anterior los términos relacionados con la oscilación se obtendrá un movimiento rotatorio puro. Esta es en realidad la única modificación que se va a realizar a la ya existente condición de frontera oscilatoria aunque el proceso para la creación de la nueva conlleva otras actuaciones adicionales que se describen a continuación.

El primer paso es crear una carpeta con el nombre de la nueva condición de frontera: *fixedAngularVelocity*. Dentro de dicha carpeta se deben copiar los archivos *.C* y *.H* de la condición de frontera que se quiere modificar y se les debe cambiar el nombre por el de la nueva en todos los lugares donde aparezca. Obviamente también se deben de realizar las modificaciones de código que sean convenientes, que en este caso se trata de convertir la ecuación citada préviamente en la siguiente:

```
scalar angle = angle0_ + (omega_*t.value());
```

Código 21. Aspecto de la parte modificada.

Además del seno, también se ha eliminado la variable *amplitude* ya que no se pretende utilizar. De este modo se deberá borrar en la declaración de la condición de frontera la definición de *amplitude* y dejar el resto como estaba.

A continuación se crea un nuevo directorio con nombre aleatorio, por ejemplo *userBCs*, dentro del cual debe de estar la carpeta que se ha creado con anterioridad y otra nueva llamada *make*. Dentro de *make* deben haber dos archivos llamados *options* y *files*. Se trata de dos archivos que se necesitan para llevar a cabo una correcta compilación y deben tener el aspecto que se muestra a continuación.

Files indica que archivo se va a compilar y donde se va a ubicar el resultado:

```
fixedAngularVelocity/fixedAngularVelocityPointPatchVectorField.C
LIB = $(FOAM_USER_LIBBIN)/libUserBCs
```

Código 22. Aspecto del archivo *files*.

Options añade los archivos de cabecera (*.H)* necesarios despúes de *EXE_INC* y incluye las librerías requeridas para la compilación después de *LIB_LIBS* :

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude
LIB_LIBS = \
    -lfiniteVolume
```

Código 23. Aspecto del archivo *options*.

Una vez se haya realizado todo lo anterior ya se está en disposición de compilar la nueva condición de frontera. Para ello se debe acceder mediante el terminal a la carpeta que engloba a *fixedAngularVelocity* y a *make -UserBCs* en este caso- y ejecutar *wmake libso*.

A partir de este momento, si la compilación se lleva a cabo sin errores, ya se dispone de la nueva condición y lo único que hace falta es incluirla en el *controlDict* para que se tenga en cuenta a la hora de simular. En este caso lo que hay que añadir a *controlDict* es lo siguiente:

```
libs ("libUserBCs.so");
```

Código 24. Código a añadir en *controlDict* para activar la nueva condición.

8.6 ROTACIÓN DEL OBSTÁCULO: INTERPOLACIONES Y REGENERACIÓN DE MALLA

Gracias a la condición de frontera creada se puede simular una rotación pura del obstáculo añadido al caso *cavity*. Se procede a su simulación durante un limitado periodo de tiempo para analizar el comportamiento de la malla.

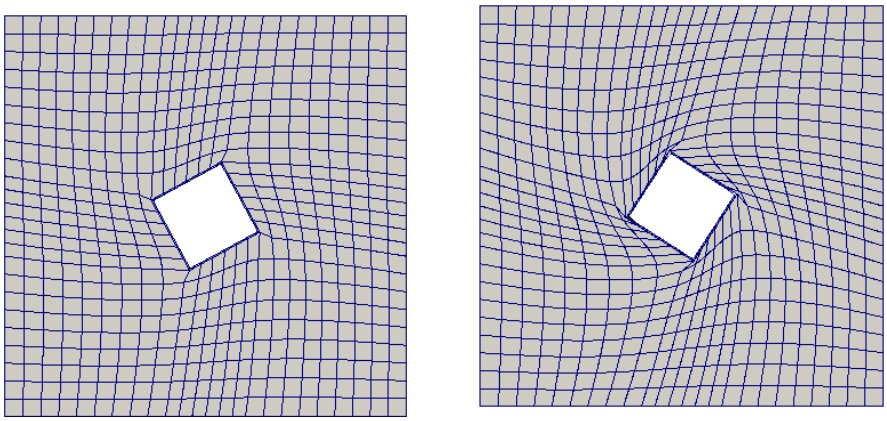


Figura 27. Movimiento de la malla en el caso *cavity* con un obstáculo en rotación.

Como se puede observar en la figura 27, es imposible que se pueda simular un giro de 360º mediante el tratamiento de malla que se está realizando. Al pasar de los 45º la malla pronto presenta errores que interrumpen el proceso de cálculo. Por esta razón, se plantea llegado este punto del estudio otras formas de manipular los puntos de la malla que permitan realizar rotaciones completas sin inducir ningún tipo de errores. Dentro de las alternativas que se han mencionado previamente en este documento, parece que la única solución sería dejar de lado el movimiento automático de la malla e implementar cambios topológicos. Más concretamente el objetivo es intentar modificar la conectividad de los puntos de la malla para permitir que los que pertenecen al obstáculo puedan rotar en sentido contrario a él evitando grandes deformaciones. Se trata de implementar el modificador topológico conocido como interfaz deslizante, del que ya se ha hablado en el apartado 7.

En los tutoriales de OpenFOAM son muy pocos los casos que tratan simulaciones que incluyan movimiento de malla con cambios topológicos. Todos los ejemplos que contienen cambios topológicos no muestran deformación de malla y esto hace que sea muy complicado preparar un caso para poder incluir dichos cambios. Además, como se ha debatido en la sección malla dinámica, no hay ninguna alternativa que satisfaga el objetivo del estudio debido al contacto entre los dos engranajes. De modo que se debe de encontrar alguna forma de obtener un giro completo sin que la malla resulte inválida.

La opción que se plantea consiste en crear mallas nuevas cada cierto periodo de tiempo para reducir la pérdida de calidad que induce el giro. Por el hecho de tratarse de una rotación con velocidad angular constante, la posición de los cuatro vértices del cuadrado es conocida para cualquier instante determinado. Esto permite generar una malla para cualquier instante de tiempo deseado, ya que los puntos a introducir en el *blockMeshDict* se conocen de antemano. Entonces, cuando el cuadrado haya rotado un ángulo en el que la malla empieze a parecer forzada, se puede introducir una malla nueva que no estará sometida a la deformación de la anterior. De esta manera se puede ir substituyendo las mallas antiguas por nuevas que tendrán capacidad para soportar giros mayores y significarán un aumento de calidad.

Se dividirán los 360º en varias partes en las que la malla será diferente. En cada transición, se debe crear la malla nueva, interpolar los resultados de la malla antigua a la nueva y ejecutar la simulación hasta el punto en el que se lleve a cabo el siguiente cambio. Para que haya una cierta continuidad se ha creado un pequeño programa que se ocupa de realizar automáticamente los pasos mencionados anteriormente para cada una

de las divisiones de los 360º. Cada una de las partes quedará guardada en una carpeta como si fuera un caso aparte, sin relación con los demás. De hecho, para OpenFOAM, cada parte será un caso independiente con la particulariedad de que tendrá unos ciertos valores definidos en la carpeta temporal inicial provinientes de una interpolación en lugar de una imposición por parte del usuario. Finalmente, se pueden agrupar todos los resultados para observarlos de forma conjunta y dar más sentido al hecho de que todos juntos forman una sola simulación.

8.6.1 EL SCRIPT

El programa que se ocupa de controlar las diferentes etapas de la simulación se engloba en lo que es conocido como un *shell script*. Se trata de un archivo de texto que está destinado a ser interpretado por la terminal de comandos del sistema operativo. En él se definen cambios de carpetas, ejecuciones de aplicaciones de OpenFOAM, eliminaciones de archivos, escritura en archivos, lectura de archivos y todo tipo de acciones necesarias para llevar a cabo la simulación. El código utilizado es exactamente el mismo que se utilizaría para realizar dichas opciones directamente en el terminal, hecho que facilita su desarrollo. A continuación se muestra la parte del *script* que se ocupa de ejecutar la segunda parte del proceso para ilustrar más detalladamente sus funcionalidades. El texto azul son comentarios incluidos para facilitar la localización y entendimiento de las diferentes partes del código. Se han incluído porque se desea mostrar el archivo original.

```
#------------------------------------------------------------------------#
#PARTE 2
echo
echo PART 2
echo
#Guardar en variables (text1, text2...) las coordenadas de los puntos que se
#han desplazado y son necesarios para el blockMeshDict. Interesan los vertices
#del cuadrado que gira y sus proyecciones en las fronteras para saber en que
#linea del archivo "points" está cada uno se buscan en
#constant/polyMesh/points. Vertices del cuadrado:

        pi=$(echo "scale=9;4*a(1)" |bc -l) #numero pi
        r=$(echo "scale=9;sqrt(0.0002)" |bc -l) #radio de los vértices
        x1=$(echo "scale=9;0.05+($r*c((5*$pi/4)+0.2))" |bc -l) #cordenadas x e
y de 0.04 0.04 al acabar primera parte
        y1=$(echo "scale=9;0.05+($r*s((5*$pi/4)+0.2))" |bc -l)
        x2=$(echo "scale=9;0.05+($r*c((7*$pi/4)+0.2))" |bc -l) #cordenadas x e
y de 0.06 0.04 al acabar primera parte
        y2=$(echo "scale=9;0.05+($r*s((7*$pi/4)+0.2))" |bc -l)
 x3=$(echo "scale=9;0.05+($r*c((3*$pi/4)+0.2))" |bc -l) #cordenadas x e
```

```
y de 0.04 0.06 al acabar primera parte
        y3=$(echo "scale=9;0.05+($r*s((3*$pi/4)+0.2))" |bc -l)
        x4=$(echo "scale=9;0.05+($r*c((1*$pi/4)+0.2))" |bc -l) #cordenadas x e
y de 0.06 0.06 al acabar primera parte
        y4=$(echo "scale=9;0.05+($r*s((1*$pi/4)+0.2))" |bc -l)
```

Código 25. Parte del *script* utilizado para la simulación de la rotación del obstáculo en *cavity*.

La parte superior se ocupa de calcular las coordenadas de los cuatro vértices del cuadrado. Se parte de las coordenadas del punto central y se le añade la variación producida por el giro desde su posición original a través de la distancia entre los vértices y el centro -radio- y el uso del número pi para los ángulos.

El extracto siguiente coloca en variables las coordenadas obtenidas previamente y las distribuye también en la coordenada de la profundidad -z-.

```
    text1="($x1 $y1 0)" # 0.04 0.04 0

    text2="($x1 $y1 0.01)" #0.04 0.04 0.01
    text3="($x2 $y2 0)" #0.06 0.04 0
    text4="($x2 $y2 0.01)" #0.06 0.04 0.01
    text5="($x3 $y3 0)" #0.04 0.06 0
    text6="($x3 $y3 0.01)" #0.04 0.06 0.01
    text7="($x4 $y4 0)" #0.06 0.06 0
    text8="($x4 $y4 0.01)" #0.06 0.06 0.01
```

Código 26. Parte del *script* utilizado para la simulación de la rotación del obstáculo en *cavity*.

Los puntos de la frontera son necesarios porque mediante blockMesh la malla se crea a partir de bloques -hexaedros- que se crearán usando los puntos de los vértices y sus correspondientes en las fronteras. Se introducen en variables primero los puntos pertenecientes al suelo y al techo del dominio.

```
#Puntos en suelo/techo de la frontera: Se cogen las coordenadas horizontales de
las variables anteriores y se le añade la coordenada "y" y "z"

    text9="($x1"" "0" ""0)" #0.04 0 0
    text10="($x1"" "0" ""0.01)" # 0.04 0 0.01
    text11="($x2"" "0" ""0)" #0.06 0 0
    text12="($x2"" "0" ""0.01)" #0.06 0 0.01
    text13="($x3"" "0.1" ""0)" #0.04 0.1 0
    text14="($x3"" "0.1" ""0.01)" #0.04 0.1 0.01
    text15="($x4"" "0.1" ""0)" #0.06 0.1 0
    text16="($x4"" "0.1" ""0.01)" #0.06 0.1 0.01
```

Código 27. Parte del *script* utilizado para la simulación de la rotación del obstáculo en *cavity*.

A continuación se hace lo mismo con los puntos de las paredes laterales del dominio.

```
#Puntos en laterales de la frontera del dominio: Se cogen las coordenadas
verticales de las variables anteriores y se le añade la coordenada "x" y
"z"

    text17="(0"" "$y1" ""0)" #0 0.04 0
    text18="(0"" "$y1" ""0.01)" #0 0.04 0.01
    text19="(0"" "$y3" ""0)" #0 0.06 0
    text20="(0"" "$y3" ""0.01)" #0 0.06 0.01
    text21="(0.1"" "$y2" ""0)" #0.1 0.04 0
    text22="(0.1"" "$y2" ""0.01)" #0.1 0.04 0.01

    text23="(0.1"" "$y4" ""0)" #0.1 0.06 0
    text24="(0.1"" "$y4" ""0.01)" #0.1 0.06 0.01
```

Código 28. Parte del *script* utilizado para la simulación de la rotación del obstáculo en *cavity*.

El siguiente paso es modificar el blockMeshDict reemplazando los puntos de la malla antigua por los obtenidos anteriormente en el código mediante el comando *sed*. Para conocer la línea en la que se deben introducir cada coordenada se debe consultar el archivo *points* de la malla original -la de la primera parte-.

```
#Cambiamos a la carpeta constant/polyMesh, donde se modificará el blockMeshDict
cd ../part2/constant/polyMesh/

#A continuación se modifican lineas en blockMeshDict para cambiar las
coordenadas de los puntos de interés que se han desplazado.
#En part1/constant/polyMesh/blockMeshDict se localizan las lineas de los puntos
que se van a modificar en el siguiente código:

     sed -i '55c '"$text1"'' blockMeshDict #Inserta "text1" en la linea 55 de
blockMeshDict, eliminando lo que había antes.
    sed -i '72c '"$text2"'' blockMeshDict
    sed -i '56c '"$text3"'' blockMeshDict
    sed -i '73c '"$text4"'' blockMeshDict
    sed -i '59c '"$text5"'' blockMeshDict
    sed -i '76c '"$text6"'' blockMeshDict
    sed -i '60c '"$text7"'' blockMeshDict
    sed -i '77c '"$text8"'' blockMeshDict
    sed -i '51c '"$text9"'' blockMeshDict
    sed -i '68c '"$text10"'' blockMeshDict
    sed -i '52c '"$text11"'' blockMeshDict
    sed -i '69c '"$text12"'' blockMeshDict
    sed -i '63c '"$text13"'' blockMeshDict
    sed -i '80c '"$text14"'' blockMeshDict
    sed -i '64c '"$text15"'' blockMeshDict
    sed -i '81c '"$text16"'' blockMeshDict
    sed -i '54c '"$text17"'' blockMeshDict
    sed -i '71c '"$text18"'' blockMeshDict
```

```
    sed -i '58c '"$text19"'' blockMeshDict

    sed -i '75c '"$text20"'' blockMeshDict
    sed -i '57c '"$text21"'' blockMeshDict
    sed -i '74c '"$text22"'' blockMeshDict
    sed -i '61c '"$text23"'' blockMeshDict
    sed -i '78c '"$text24"'' blockMeshDict
cd ..
cd ..
```

Código 29. Parte del *script* utilizado para la simulación de la rotación del obstáculo en *cavity*.

Una vez blockMeshDict está listo ya se puede crear la malla de la segunda parte ejecutando blockMesh. Después, se interpolan los datos de la última carpeta temporal de la parte anterior -la primera en este caso- mediante *mapFields* para poder reanudar la simulación justo donde se había interrumpido.

```
#Se ejectuta blockMesh en part2 con los nuevos puntos
    echo
    runApplication blockMesh
echo

#Se interpolan los resultados de la última iteración de part1 a la nueva malla de
part2
echo "mapping fields from part1"
    mapFields ../part1 -sourceTime latestTime > log.mapFields 2>&1
```

Código 30. Parte del *script* utilizado para la simulación de la rotación del obstáculo en *cavity*.

Se debe de eliminar un archivo -*pointMotionU.unmapped*- porque se crea durante la interpolación y no es necesario. Aparece porque *mapFields* -la herramienta para las interpolaciones- no es capaz de interpolarlo. Dicho archivo se copia de la última carpeta temporal de la parte anterior -part1-. Así, la segunda parte ya está lista para reanudar la simulación mediante la ejecución de *icoDyMFoam*.

```
#El archivo pointMotionU no es interpolado por mapFields. Se copia el último de
la parte anterior (part1)

    cp -t 0.004 ../part1/0.004/pointMotionU

#Por el hecho de no interpolar pointMotionU se crea pointMotionU.unmapped, que
como no interesa, se elimina

    cd 0.004/
    rm -r pointMotionU.unmapped
    cd ..
#Se ejecuta icoDyMFoam en part2
```

```
echo
 runApplication $application
```

Código 31. Parte del *script* utilizado para la simulación de la rotación del obstáculo en *cavity*.

Las siguientes partes de la rotación completa -360º- siguen la misma estructura variando únicamente el nombre de las carpetas temporales a las que se accede y las coordenadas de los nuevos puntos.

8.6.2 REMALLADO Y CAMBIO DE BLOQUES

La creación de nuevas mallas aporta al tratamiento elegido la posibilidad de ser utilizado en varias partes para crear una rotación completa. Cada vez que se continúa la simulación en una nueva malla se mejora la calidad de ésta y se aumenta el ángulo de giro que se podría llevar a cabo sin errores. Sin embargo este método también tiene sus inconvenientes, principalmente el hecho de que se debe de realizar una interpolación entre mallas. Este proceso no está libre de errores y puede empeorar en cierta medida la calidad de los resultados obtenidos.

El problema más importante se observa cuando ya se han realizado varias partes de la rotación completa. Lo que ocurre es que se observa que a medida que la simulación avanza los bloques que parten de los cuatro lados del cuadrado en rotación van reduciendo su anchura, es decir, la proyección de cada uno de los lados sobre la frontera del dominio. Obviamente dicha proyección se reducíria a un punto si no se incorporara ninguna modificación al *script* que redefiniera la formación de los hexaedros base de la malla. Para evitar la reducción de los bloques se debe de realizar una modificación al *blockMeshDict*.

Supóngase que se tiene la situación de la figura 28. Cuando el obstáculo ha girado 45º, la proyección de cualquier cara sobre los dos lados de la frontera a los que apunta es la misma. En el caso de la figura, las proyecciónes 1 y 2 tienen la misma anchura. Sin embargo, si el giro antihorario continúa, la proyección 2 disminuirá progresivamente mientras que la 1 aumentará. La proyección 2, que es la que se ha utilizado desde que el obstáculo estaba con ángulo de inclinación 0º debe de ser corregida ya que ha quedado demostrado que a partir de este punto es mejor para la calidad de la malla que se utilize la proyección 1.

El cambio de bloques permite continuar con la simulación sin que la malla sufra errores graves. Cabe destacar que la modificación se lleva a cabo por lo que ocurre al proseguir

con la rotación ya que en el instante en el que el giro es 45º, la calidad de malla será la misma independientemente de como esté definido el bloque.

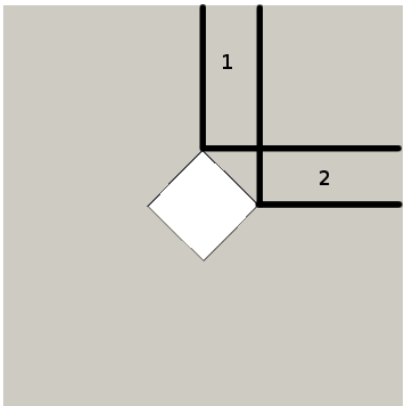
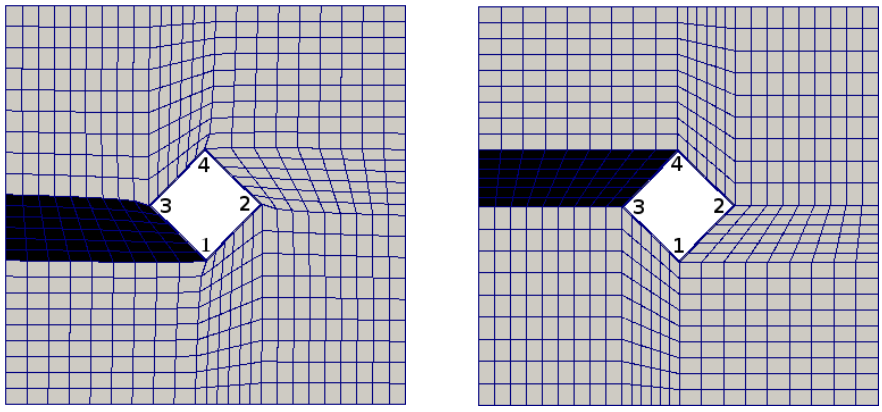


Figura 28. Posición del obstáculo a los 45º de rotación. Se muestran las proyecciones de una de las caras sobre las fronteras a las que enfoca: números 1 y 2.

La figura 29 ilustra la generación de una nueva malla después de haber ejecutado una parte de la rotación. Se trata exactamente del cambio de bloques que se lleva a cabo a los primeros 45º de giro.



a *b*

Figura 29. Posición del obstáculo a los 45º de rotación. Se muestra la numeración de vértices utilizada en el *script*. La zona sombreada corresponde a uno de los bloques de la malla. En *a* la situación es previa al cambio de bloques; en *b* es posterior .

Las partes en las que se realize un cambio de bloques no necesitan realizar ninguna modificación en el *blockMeshDict*. Lo único que se debe de variar es la parte del *script*

en la que se definen las variables “text” en función de las coordenadas horizontales y verticales de los vértices del cuadrado. Básicamente se trata de una redefinición de los vértices del obstáculo que consiste en adjudicar las coordenadas de un cierto punto al inmediatemente siguiente en sentido horario. El resultado es un deslizamiento de los bloques de un lado del obstáculo al contiguo moviéndose en sentido horario. La figura 29 muestra como el bloque sombreado desplaza sus puntos de contacto a los inmediatamente consecutivos en sentido horario.

Para mostrar el efecto del cambio en el código del *script* se incluye a continuación un pequeño extracto de la declaración de las primeras 8 variables antes de haber realizado el cambio de bloques.

```
text1="($x1 $y1 0)" # 0.04 0.04 0

text2="($x1 $y1 0.01)" #0.04 0.04 0.01
text3="($x2 $y2 0)" #0.06 0.04 0
text4="($x2 $y2 0.01)" #0.06 0.04 0.01
text5="($x3 $y3 0)" #0.04 0.06 0
text6="($x3 $y3 0.01)" #0.04 0.06 0.01
text7="($x4 $y4 0)" #0.06 0.06 0
text8="($x4 $y4 0.01)" #0.06 0.06 0.01
```

Código 32. Extracto del *script* utilizado para la simulación de la rotación del obstáculo en *cavity*.

A continuación se muestra el mismo trozo de código pero con los cambios que se necesitan para el deslizamiento de los bloques.

```
text1="($x3 $y3 0)" # 0.04 0.04 0

text2="($x3 $y3 0.01)" #0.04 0.04 0.01
text3="($x1 $y1 0)" #0.06 0.04 0
text4="($x1 $y1 0.01)" #0.06 0.04 0.01
text5="($x4 $y4 0)" #0.04 0.06 0
text6="($x4 $y4 0.01)" #0.04 0.06 0.01
text7="($x2 $y2 0)" #0.06 0.06 0
text8="($x2 $y2 0.01)" #0.06 0.06 0.01
```

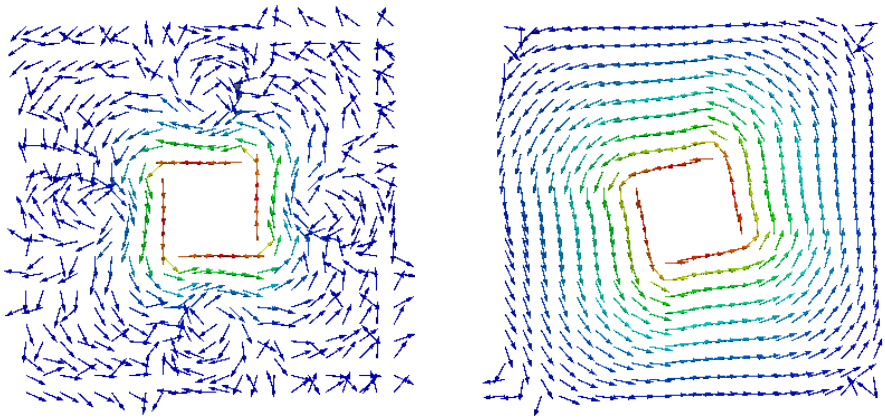
Código 33. Extracto del *script* utilizado para la simulación de la rotación del obstáculo en *cavity*.

El cambio en el orden de inserción de coordenadas refleja el desplazamiento horario de los bloques. Por ejemplo, las variables *text1* y *text2* pertenecían al vértice 1 antes del cambio, mientras que ahora representan al 3. Realizando lo mismo en el resto de variables, *blockMeshDict* no hace falta que sea modificado.

8.6.3 INTERPOLACIONES

Gracias a las interpolaciones, cada vez que se realiza un cambio de malla se puede continuar con la simulación donde se había interrumpido. Los valores de los campos de presión y velocidad se transfieren a la nueva malla permitiendo que se consiga continuidad en los resultados. Sin embargo, el fichero que se ocupa del movimiento de la malla -*pointMotionU*- no es interpolado por OpenFOAM y se debe copiar directamente de la última carpeta temporal de la parte anterior.

Al empezar la simulación, con el obstáculo quieto y en la posición original, el fluido parte del reposo y le cuesta adaptar la circulación alrededor del obstáculo. En cambio, al principio de la segunda parte, una vez interpolados los resultados, la circulación ya está presente -ver figura 30-.



a *b*

Figura 30. Campo fluido del caso *cavity* con un obstáculo en rotación. En *a* se muestra el inicio de la primera parte de la rotación; en *b* el inicio de la segunda.

La adaptación del fluido a la circulación producida por la rotación se produce de forma bastante rápida debido a que la viscosidad del fluido es elevada. Como se ha mencionado anteriormente, el objetivo es implementar el giro en un engranaje. Además, dicho engranaje se utilizará en una turbo-bomba de aceite, razón por la que se ha impuesto una viscosidad elevada en el fluido. Las consecuencias se muestran en la figura 30, donde se ve como prácticamente desde el primer instante la zona más cercana al obstáculo ya ha adoptado la dirección de la rotación.

8.7 ROTACIÓN EN MALLAS MÁS FINAS

Mediante el proceso planteado en este apartado se puede simular la rotación del cuadrado sin ningún límite geométrico. Para analizar más detalladamente el movimiento de la malla se puede realizar lo mismo pero con mallas más finas. Así se puede estudiar, mediante comparación, el coste computacional de la deformación de los puntos para ver la influencia de su densidad en el tiempo de la simulación.

A continuación, se desea repetir la simulación de la rotación del obstáculo en el caso *cavity* pero ahora con una malla más fina. Previamente se había utilizado una malla dividida en 576 celdas y se desea aumentar ese número a 2916, un poco más de 5 veces mayor. Una malla con más puntos es capaz de calcular con más detalle el comportamiento del fluido en el dominio, pero hay que tener en cuenta que añade un cierto coste computacional. Mientras una simulación con la malla pequeña dura como mucho 26 segundos, en el caso de la malla grande la duración sobrepasa los 2 minutos. Para ilustrar mejor la influencia de la densidad de la malla se incluye la tabla 1 que indica el tiempo de ejecución -mediante *icoDyMFoam*- de las 8 primeras partes de la rotación del cuadrado, que corresponden a un giro de 90º. La única variación entre ambas es el incremento en el número de puntos de la malla, el resto de parámetros se han dejado intactos para poder obtener una comparación válida.

	576 celdas	**1156 celdas**	**Razón**	**2916 celdas**	**Razón**
Parte 1	24 s	45 s	x1.8	123 s	x5.1
Parte 2	25 s	49 s	x1.9	132 s	x5.2
Parte 3	25 s	50 s	x2	138 s	x5.5
Parte 4	26 s	49 s	x1.8	139 s	x5.3
Parte 5	26 s	50 s	x1.9	144 s	x5.5
Parte 6	25 s	49 s	x1.9	139 s	x5.5
Parte 7	25 s	48 s	x1.9	137 s	x5.4
Parte 8	24 s	47 s	x1.9	140 s	x5.8
Total	200 s	387 s	x1.9	1092 s	x5.4

Tabla 1. Tiempo de ejecución de *icoDyMFoam* para cada una de las partes de la rotación en función de tres densidades de malla. Se muestra también la razón respecto a la menos densa. Procesador utilizado: *Intel Pentium M 1.73GHz*.

Se observa en los resutados de la tabla 1 que, para la rotación del obstáculo en el caso *cavity*, incrementar el número de celdas comporta un aumento de aproximadamente la misma proporción en el tiempo de ejecución. Esto se debe de tener en cuenta ya que no interesa generar mallas demasiado densas que conlleven una simulación muy lenta si no

mejoran la calidad de los resultados de forma considerable.

Se observan variaciones del tiempo de ejecución entre las diferents partes de una misma malla. Además, cuanto mayor es el tamaño de la malla, mayor es la diferencia. Llegado este punto interesa saber si la causa de este fenómeno es el tratamiento de la malla o el del fluido. Es posible que cuando la malla resulte forzada, el *solver* necesite más tiempo para manipular su movimiento. Además, puede que en ese caso la resolución del campo fluido sea más lenta de lo normal. Para salir de dudas, se ejecutará una aplicación que simula únicamente el movimiento de los puntos de la malla: *moveMesh*. En función de los tiempos de ejecución obtenidos se podrá saber si la deformación de la malla causa lentitud en la simulación. Obviamente, se espera que los tiempos sean menores que los de la tabla de *icoDyMFoam* ya que en este caso se realizan menos cálculos.

	576 celdas	**1156 celdas**	**2916 celdas**
Parte 1	14 s	29 s	76 s
Parte 2	15 s	29 s	76 s
Parte 3	15 s	28 s	77 s
Parte 4	14 s	28 s	76 s
Parte 5	14 s	28 s	78 s
Parte 6	14 s	28 s	77 s
Parte 7	15 s	29 s	77 s
Parte 8	14 s	29 s	76 s

Tabla 2. Tiempo de ejecución de *moveMesh* apara cada una de las partes de la rotación en función de tres densidades de malla. Procesador utilizado: *Intel Pentium M 1.73GHz*.

En los valores de la tabla 2 se puede observar que, como era de esperar, tratando únicamente el movimiento de la malla el tiempo de ejecución es significativamente menor. Es de especial importancia destacar que para todas las partes de la rotación es prácticamente idéntico. Para el caso de mallas pequeñas coincide con la variación de la simulación total -malla y fluido-. Sin embargo, en mallas grandes, las variaciones son de una magnitud mucho menor a las que aparecen en la simulación con el *solver* completo -malla y fluido-. Por lo tanto, el hecho de que para una misma malla, de relativamente alta densidad, las simulaciones de diferentes partes sean más, o menos, lentas se atribuye a las complejidades que aparecen en el flujo.

Es importante destacar que en todos los tamaños de malla, el tiempo de cálculo de los puntos de la malla ocupa un 50% - 60% del tiempo total de la simulación. Esto demuestra el esfuerzo computacional que se realiza durante ese proceso y el consiguiente aumento de tiempo de una simulación con movimiento automático de malla.

Resulta interesante analizar los tiempos de tratamiento de malla en función del tipo de difusividad aplicado para ver su influencia en la simulación completa. Las tablas 3, 4 y 5 muestran, para cada uno de los tres tamaños de malla analizados, los tiempos de ejecución para cada una de las 8 primeras partes de la rotación con tres tipos diferentes de distribuciones de difusividad. Como cabe esperar, los datos de las tablas reflejan que el sistema más simple es el más rápido y viceversa. De modo que la distribución de difusividad según el inverso de la distancia, pese a producir mejores resultados, es el más costoso de los tres computacionalmente.

	direccional	**Distancia inversa**	**Constante**
Parte 1	14 s	16 s	15 s
Parte 2	15 s	16 s	14 s
Parte 3	15 s	16 s	14 s
Parte 4	14 s	17 s	15 s
Parte 5	14 s	16 s	14 s
Parte 6	14 s	16 s	14 s
Parte 7	15 s	16 s	15 s
Parte 8	14 s	16 s	14 s

Tabla 3. Tiempos de ejecución de *moveMesh* para cada una de las partes de la rotación en la malla de 576 celdas. En cada columna se encuentran los resultados para diferentes esquemas de difusividad. Procesador utilizado: *Intel Pentium M 1.73GHz*.

	direccional	**Distancia inversa**	**Constante**
Parte 1	29 s	29 s	28 s
Parte 2	29 s	30 s	28 s
Parte 3	28 s	31 s	27 s
Parte 4	28 s	33 s	28 s
Parte 5	28 s	33 s	28 s
Parte 6	28 s	32 s	28 s
Parte 7	29 s	31 s	27 s

Parte 8	29 s	29 s	27 s

Tabla 4. Tiempos de ejecución de *moveMesh* para cada una de las partes de la rotación en la malla de 1156 celdas. En cada columna se encuentran los resultados para diferentes esquemas de difusicidad. Procesador utilizado: *Intel Pentium M 1.73GHz*.

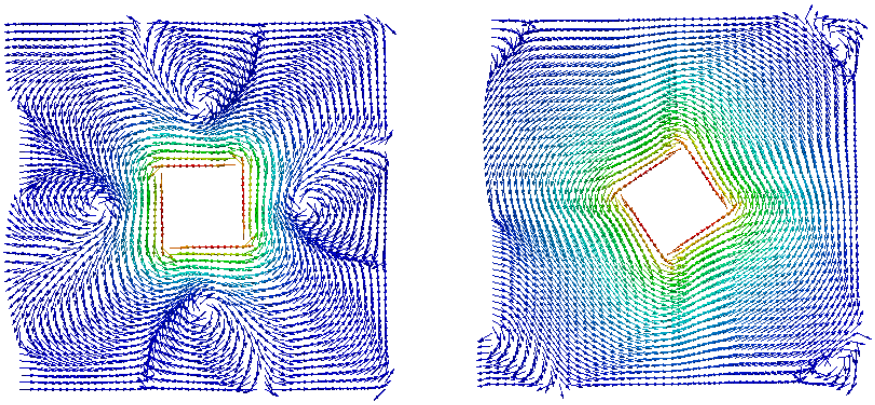
	direccional	**Distancia inversa**	**Constante**
Parte 1	76 s	82 s	75 s
Parte 2	76 s	84 s	75 s
Parte 3	77 s	87 s	76 s
Parte 4	76 s	93 s	77 s
Parte 5	78 s	92 s	78 s
Parte 6	77 s	88 s	76 s
Parte 7	77 s	83 s	76 s
Parte 8	76 s	82 s	76 s

Tabla 5. Tiempos de ejecución de *moveMesh* para cada una de las partes de la rotación en la malla de 2916 celdas. En cada columna se encuentran los resultados para diferentes esquemas de difusicidad. Procesador utilizado: *Intel Pentium M 1.73GHz*.

Por otro lado, se observa que para mallas más finas el tiempo sufre ciertas variaciones a medida que avanzan las partes de la rotación. Además, el margen es mucho mayor en el caso de la difusividad inversamente proporcional a la distancia. El tiempo de ejecución para un mismo esquema de difusividad alcanza su máximo en las partes 4 y 5, que son las más cercanas al punto en el que se ha rotado 45º. Puede que el incremento en el tiempo de ejecución sea debido a que en ese momento de la simulación es cuando la malla está más forzada y deformada. Sin embargo, en la malla menos densa el tiempo permanece prácticamente inalterado a lo largo de todas las partes.

8.7.1 RESULTADOS

Se decide simular la rotación con la malla de 2916 celdas para observar las diferencias respecto al caso de 576. El proceso es exactamente el mismo que el que se ha detallado en la sección 8.6. Por suerte, los puntos clave para crear la malla son los mismos y la única diferencia reside en el número de celdas que se deben de incluir en el *blockMeshDict*.

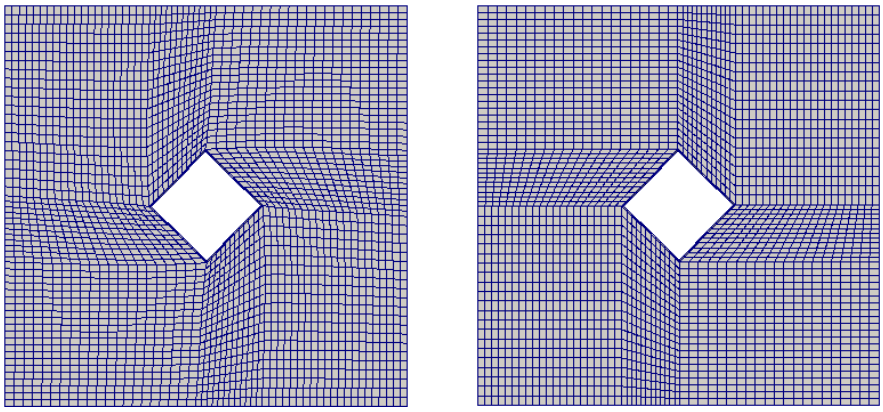


a *b*

Figura 31. Campo de velocidades del caso *cavity* con un obstáculo en rotación con una malla de 2916 celdas. En *a* se muestra el inicio de la primera parte de la rotación y en *b* el de la cuarta.

La figura 31 incluye algunos momentos de la simulación demostrando como el incremento en la densidad de la malla permite obtener unos resultados más precisos. Se observan detalladamente circulaciones del fluido que en el caso de menor densidad de malla -figura 30- prácticamente no se observan.

En esta caso también es necesario realizar el cambio de bloques para asegurar la continuación sin errores de la rotación del obstáculo. Se observa en este caso -ver figura 32- como la malla no parece sufrir tanto como en el caso con un menor número de celdas.



a *b*

Figura 32. Posición del obstáculo a los 45º de rotación. Se muestra el efecto del cambio de bloques pasando de la situación de *a* a la de *b*.

8.7.2 RESULTADOS CON ADICIÓN DE FLUJO HORIZONTAL

Una configuración que puede resultar de cierto interés es el hecho de añadir un flujo horizontal uniforme en todo el dominio con una velocidad similar a la que el obstáculo aplica al fluido debido a la rotación. Se pretende observar como la circulación provocada por la rotación del cuadrado influye en la corriente incorporada. La curiosidad de este caso reside exclusivamente en el comportamiento del fluido, ya que la malla realizará exactamente el mismo movimiento que en los casos presentados previamente. Por esta razón se escoge la malla más densa, que permitirá representar los resultados con mayor precisión.

Para configurar la nueva característica del flujo se debe de modificar aspectos de la malla y de las condiciones de frontera. En las simulaciones sin flujo horizontal, todas las paredes de la frontera exterior del dominio pertenecían a un mismo tipo de condición de frontera en la que la velocidad era nula. Ahora las fronteras laterales deben ambas cambiar sus características. Por esta razón en el *blockMeshDict* se deben crear dos *patch* nuevos a los que se asignarán la entrada y la salida de fluido del dominio. Al izquierdo se le adjudicará una velocidad horizontal positiva de 0.25 m/s -sentido hacia la derecha- y al derecho se dejará que la propia simulación obtenga su velocidad. No se puede forzar que la salida tenga la misma velocidad que la entrada porque el valor en ese punto debe de satisfacer la continuidad.

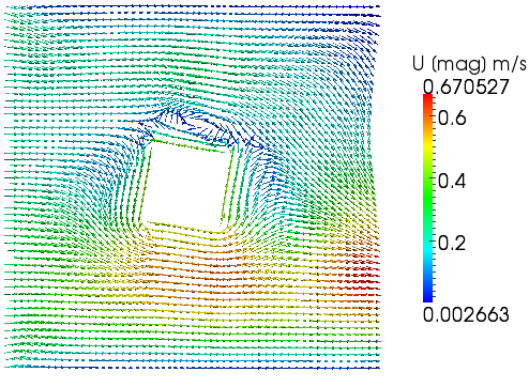


Figura **33**. Campo de velocidades del caso *cavity* con obstáculo en rotación y flujo horizontal añadido. Los valores de la escala corresponden al módulo del vector velocidad.

El resultado esperado es una deceleración del fluido en la parte superior del obstáculo y una aceleracíon -por encima de la velocidad lineal en el cuadrado- en la inferior. Se trata simplemente de superponer la circulación que se ha mostrado en el caso anterior con una corriente horizontal de izquierda a derecha. La figura 33 ilustra como el comportamiento

del fluido encaja con lo lógicamente esperado. La velocidad de la corriente a la entrada es de 0.25 m/s, y la velocidad lineal en el obstáculo debido a la rotación 0.5 m/s.

Como se ha demostrado anteriormente, las complicaciones en el fluido son la principal causa de prolongamiento del tiempo de ejecución en mallas finas, por lo que se espera que en este caso los tiempos de ejecución sean superiores a los del caso que incluye únicamente rotación. Para demostrarlo se incluye en la tabla 6 una comparación de tiempos entre el caso con corriente y el que no la incluye. Como la malla se mueve de la misma forma que en el caso anterior con malla densa, no tiene interés volverlo a calcular en este caso. Se observa un aumento de tiempo de ente el 10% y el 20% respecto al caso en el que no se incluye flujo horizontal. De modo que cuanto más complicado sea el flujo a tratar, más coste computacional se le asociará.

	Sin viento	**Con viento**	**Razón**
Parte 1	123 s	152 s	x1.23
Parte 2	132 s	153 s	x1.16
Parte 3	138 s	156 s	x1.13
Parte 4	139 s	159 s	x1.14
Parte 5	144 s	158 s	x1.09
Parte 6	139 s	155 s	x1.11
Parte 7	137 s	164 s	x1.19
Parte 8	140 s	158 s	x1.12
Total	1092 s	1255 s	x1.15

Tabla 6. Tiempos de ejecución de *icoDyMFoam* para cada una de las partes de la rotación en la malla de 2916 celdas. Se muestran los resultados para el caso sin flujo horizontal y con él. En la última columna se muestra la relación del último respecto al primero. Procesador utilizado: *Intel Pentium M 1.73GHz*.

8.8 EJECUCIÓN EN PARALELO

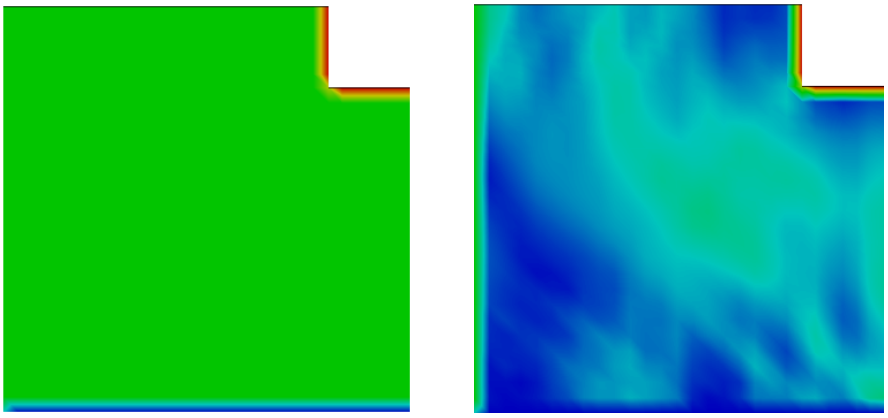
Es necesario implementar la rotación del obstáculo mediante una simulación en paralelo para reducir el tiempo de simulación al mínimo. Manteniendo los engranajes como objetivo, su malla es considerablemente más densa y el flujo más complicado debido a la geometría. En consecuencia, el tiempo de simulación puede ser de varias semanas. Una cantidad inaceptable. Consiguiendo la simulación en paralelo del obstáculo cuadrado, lo único necesario para realizar lo mismo con el engranaje es un cambio de geometría.

OpenFOAM ofrece la posibilidad de realizar simulaciones en paralelo para poder agilizar las simulaciones más lentas. El dominio completo se puede descomponer en tantas partes como procesadores se vayan a utilizar y cada una se ejecuta a parte. Los procesadores se comunican entre ellos para compartir la información de los puntos que se encuentran en las fronteras entre procesadores. En definitiva, se reparte el mismo trabajo entre varias máquinas para intentar reducir al máximo el tiempo de la simulación.

La comunicación entre procesadores añade un cierto coste computacional, por lo que existe un límite a partir del cual ésta haría que la simulación pudiera tardr lo mismo que sin la descomposición del dominio. Se debe de elegir adecuadamente el número de procesadores para optimizar la ejecución y reducir el tiempo al mínimo.

8.8.1 PREPARACIÓN DEL DOMINIO

Existe una aplicación de OpenFOAM que es la encargada de dividir el dominio en diferentes partes. Se denomina *decomposePar*, y como es típico necesita un archivo de apoyo que es el que está pensado para que modifique el usuario: *decomposeParDict*. En él se escoge entre los diferentes métodos de descomposición y sus parámetros principales, como por ejemplo el número de subdominios o el número de divisiones del dominio en horizontal, vertical y en profundidad -de interés en casos 3D-.



a *b*

Figura 34. Descomposición del caso cavity con obstáculo. Se muestra la división inferior derecha. En a sin flujo inicializado, solo se observan las condiciones de contorno y el flujo interno establecido de 0.25 m/s -verde-. En b existe un flujo inicializado a parte de las condiciones de contorno.

DecomposePar divide la malla en ciertos grupos de puntos por lo que el valor de las variables que les corresponden son también descompuestos. Del mismo modo, los archivos de las condiciones de contorno serán divididas en sus correspondientes subpartes. Cada región descompuesta del dominio recibirá una nueva carpeta denominada por el número de procesador que le corresponde, por ejemplo la del procesador 2 será *processor2*. En éstas habrá una carpeta conteniendo información sobre la malla -*constant*- y una con información sobre las variables que le pertocan según la división -*carpeta temporal inicial*- incluyendo las condiciones de contorno.

El método de descomposición utilizado recibe el nombre de *simple*. Descompone el dominio a partes iguales en función del número de divisiones que desea el usuario. Se puede elegir si la descomposición se hace en los ejes x, y o z, por ejemplo, una división del dominio en 4 partes iguales se realiza descomponiendo 2 partes en x y 2 en y. Mediante este método OpenFOAM realiza la descomposición para que los dominios cuadren exactamente con lo estipulado en las divisiones.

Una vez colocado el archivo *decomposeParDict* en la carpeta *system* ya se puede ejecutar su aplicación correspondiente para llevar a cabo la división. Por ejemplo, si se especificara que se desea dividir en cuatro partes el dominio del caso *cavity* con obstáculo, aparecerían cuatro nuevas carpetas denominadas *processor0, processor1, processor2* y *processor3*. Tanto si se tratara de una simulación que parte de 0 como si ya posee una cierta configuración de flujo establecida, *decomposePar* asigna los valores a cada una de las partes. El aspecto de una de las divisiones -con flujo inicializado y sin él- es el que muestra la figura 34.

8.8.2 PREPARACIÓN DE LA EJECUCIÓN

Una vez descompuesto con éxito el dominio, el caso ya está preparado para ser ejecutado mediante varios procesadores. Es necesario tener acceso a un ordenador con más de un procesador para poder realizar la simulación en paralelo más sencilla -dos procesadores-, sin embargo, interesa poder realizar descomposiciones en un mayor número de dominios. Gracias a los profesores David Del Campo y Roberto Castilla se ha conseguido una cuenta en el *cluster* de cálculo Mosct del Servicio Informático del Campus Terrassa, que dispone de 9 ordenadores de 4 procesadores cada uno.

Los casos se envían al *cluster* mediante Internet y se ejecutan mediante el terminal. En Mosct no está instalada la herramienta de postproceso *paraView* -por motivos de espacio- por lo que una vez finalizada la simulación el caso se debe copiar a otro

ordenador para poder analizarlo. Cabe decir que para poder llevar a cabo la simulación del caso *cavity* con obstáculo se debe añadir al *cluster* el *solver IcoDyMFoam* y la nueva condición de contorno *fixedAngularVelocity* ya que son modificaciones propias no incluidas en la distribución oficial de OpenFOAM.

Para empezar, se realiza una división en 4 dominios en la que cada una de las partes tiene un aspecto como el que se muestra en la figura 34. La descomposición se puede llevar a cabo en un ordenador con un solo procesador, pero el paso siguiente -la propia ejecución- requiere el traslado al *cluster*. Únicamente falta añadir un archivo llamado *machines* en la carpeta personal de Mosct para poder empezar la simulación. Se debe incluir en el caso de que se ejecute la simulación a través de una red, no cuando se ejecuta desde el ordenador que posee los procesadores. *Machines* es simplemente una lista de las computadoras disponibles con el número de procesadores de cada una de ellas. Solo deben de incluirse en la lista los recursos que se deban utilizar, así que para el caso de 4 subdominios solo se debe incluir un ordenador con sus cuatro procesadores.

Para ejecutar la simulación se utiliza *openMPI*. Es un protocolo que permite las comunicaciones entre procesos y es muy utilizado en situaciones que utilizan multi-procesadores. El código que se necesita para dar todas las instrucciones para la ejecución es bastante complejo. Se incluye una muestra a continuación:

```
mpirun -np 4 -hostfile /home/mf/usuaris/e3559475/machines icoDyMFoam -parallel
```

Código 34. Orden utilizada para ejecutar una simulación en paralelo-

Mpirun es la aplicación para poder realizar la ejecución en paralelo; *-np 4* indica que el número de procesadores a utilizar es 4; *-hostfile /home/mf/usuaris/e3559475/machines* se utiliza para localizar el archivo *machines*; y *icoDyMFoam -parallel* indica que se va a ejecutar el *solver* con dicho nombre en paralelo. Además, existe la opción de añadir al principio la palabra *nohup,* que permite cerrar el ordenador desde el que se ha mandado la orden -y por tanto, cerrar también su comunicación con el *cluster*- sin que la simulación se detenga. Además, si se desea que se guarde un archivo con todo el proceso se puede añadir al final *> name 2>&1*, que guardaría dicha información en el archivo *name* dentro de la carpeta desde la que se ha ejecutado la orden.

Una vez se accede a la cuenta en Mosct ya se puede realizar la simulación. Ejecutando la orden que se describe en el párrafo anterior empieza el proceso. Desafortunadamente, aparece un primer error informando que los valores del *patch movingWall* dentro de *pointMotionU* no tienen el tamaño esperado. Comparando el archivo del dominio

completo con el de un subdominio se observa que los valores de la condición de frontera *fixedAngularVelocity* no han sido descompuestos.

En este tipo de situaciones en las que hace falta tener un conocimiento más profundo de OpenFOAM para proponer alternativas coherentes es muy provechoso consultar el foro de cfd-online -[14]-. El autor de este estudio es un usuario registrado y ha participado activamente en diversos temas del foro de OpenFOAM. Según la información publicada en él, existen dificultades cuando se trata de dividir condiciones de frontera creadas por los usuarios. Así que esta podría ser la causa del problema. La alternativa para solucionar el error consiste en realizar la descomposición manualmente adjudicando a cada parte del dominio sus puntos correspondientes y eliminando el resto. Por suerte el proceso no es complicado porque la parte de *PointMotionU* que se debe de dividir son coordenadas geométricas de los puntos del obstáculo. Como la descomposición se ha realizado en partes iguales se conoce cuales son los de cada subdominio y se pueden descartar los sobrantes. Esto provoca que el tiempo dedicado a la descomposición sea significantemente mayor al de un caso que solo incluya condiciones de frontera preexistentes en la distribución del software.

Una vez finalizada la descomposición manual el caso ya está preparado para la simulación mediante 4 procesadores. Sin embargo, al comienzo de la simulación aparece otro error. Esta vez se trata de una cuestión geométrica. Hay caras del dominio que no pueden comunicarse con las vecinas porque son de un tamaño que excede su margen de tolerancia. Por alguna razón, llega un momento en la simulación en la que dos caras vecinas son demasiado diferentes y el proceso completo se detiene. Así que se decide replantear desde el momento de la descomposición para encontrar una alternativa.

Como el error esta relacionado con los tamaños de las caras en las fronteras entre procesadores, puede estar provocado por la deformación causada por el giro. Una de las posibles soluciones al problema consiste en realizar la división del domino a lo largo del eje z, en vez de en el plano x-y. De esta manera, cada procesador se ocuparía de resolver un plano x-y completo, y en el momento de comunicarse con otro, se encontraría la misma geometría y quizás desaparece el problema mencionado. Para aplicar esta idea se debe de tratar el problema de forma tridimensional, añadiendo celdas en la profundidad del domino. Obviamente las condiciones de contorno deben de modificarse también: ahora ya no existen los *patch empty*. Se realiza una prueba con una descomposición en 4 dominios en profundidad, pero vuelve a aparecer el mismo error.

Después de haber consultado numerosos documentos y comentarios en el foro de cfd-online -[14]- se adopta una nueva aproximación. En el archivo *pointMotionU* de cada

carpeta temporal inicial dentro de cada procesador, aparece siempre una lista de coordenadas en el *patch movingWall* -ver código 35-. La solución propuesta consiste en eliminarla. De este modo la descomposición manual es mucho más rápida, e incluso se ha automatizado mediante un *script*. Aunque no se tiene constancia ni evidencias de que el método funcione, se hace una prueba y la ejecución empieza y acaba sin problemas. El caso *cavity* con rotación del obstáculo cuadrado se ha simulado mediante 4 procesadores dentro de un mismo ordenador del *cluster*: *mosct02*. La división del dominio se ha realizado en el plano x-y porque ha quedado demostrado que si se hace en profundidad el tiempo de ejecución se dispara.

```
movingWall
    {
        type            fixedAngularVelocity;
        axis            (0 0 1);
        origin          (0.05 0.05 0);
        angle0          0;
        omega           50;
        p0              nonuniform List<vector>
96
(
(0.04 0.04 0)
(0.04 0.04 0.01)
[...]
(0.05833333333 0.06 0)
(0.05833333333 0.06 0.01)
)
;
        value           nonuniform 0();
    }
```

Código 35. Extracto del código de *pointMotionU* dentro de una de las carpetas *processor*. Las líneas en negrita ilustran la parte que debe de ser eliminada para la correcta simulación.

Se observa que al principio de la simulación, se pide la contraseña de acceso al ordenador *mosct02*, que es el elegido para el cálculo. Esto significa que cuando la simulación se vaya a realizar mediante más de un ordenador, cada vez que se cambie de computadora el usuario deberá introducir su contraseña. Esto no permitiría disfrutar de las ventajas del comando *nohup*, que está pensado precisamente para no tener que estar pendiente de la simulación. Para solucionar esta cuestión, se debe de hacer que los diferentes ordenadores del *cluster* conozcan sus claves y direcciones IP de modo que se consideren “conocidos” entre ellos.

Las conexiones con el *cluster*, y dentro de él, se realizan mediante *Secure Shell* o *ssh*. Se

trata de un protocolo de acceso seguro entre ordenadores que permite, entre otros, la transferencia de archivos y la ejecución de comandos mediante un terminal remoto. Para poder realizar la conexión, debe de estar instalada la aplicación en ambas partes de la conexión.

Si en cualquier ordenador preparado para realizar una conexión mediante ssh se ejecuta la orden *ssh-keygen -t rsa*, se genera un código identificativo que puede incluirse dentro de la debida carpeta -$HOME/*.ssh/*- del segundo ordenador para que lo considere como una conexión segura y no pida la contraseña. Realizar esta operación en Mosct es un poco tediosa porque se trata de varios ordenadores que comparten un único disco.

Por defecto, al conectar con *mosct.upc.es* se accede al ordenador *mosct01*. Para que el resto de ordenadores se reconcozcan entre sí basta con conectarse a uno de los 8 ordenadores restantes mediante *ssh* y ejecutar *ssh-keygen -t rsa*. La clave generada se puede enviar de nuevo a *mosct01*, por ejemplo, e introducirla dentro de un archivo llamado *authorized_keys* dentro de la carpeta $HOME/.ssh/. El paso final consiste en añadir dentro de la misma carpeta un archivo llamado *known_hosts* que contenga una lista con el nombre de cada uno de los ordenadores de Mosct, seguido de su IP y de la misma clave que se ha generado en uno de los ordenadores de Mosct.

A partir de este momento, la conexión entre ordenadores dentro del *cluster* se considerará segura y no será necesario introducir ninguna contraseña. De este modo se podrá lanzar una orden mediante *nohup* y salir de la sesión sin ninguna repercusión en el proceso de cálculo.

8.8.3 EJECUCIÓN Y POSTPROCESO CON MÚLTIPLES PROCESADORES

Ejecutando la orden especificada en la sección anterior empieza la simulación en 4 procesadores dentro de uno de los ordenadores de Mosct. Los resultados se guardan en carpetas temporales como si se tratara de un caso ejecutado en un solo procesador, con la diferencia de que estas carpetas aparecen únicamente dentro de cada una de las carpetas *processor*. Con esta organización de datos, no es posible visualizar los resultados del dominio completo, solo se puede acceder a los resultados de cada subdominio por separado -figura 35-.

Para poder extraer los datos de cada procesador y juntarlos para formar el dominio completo se utilizan las aplicaciones *reconstructParMesh* y *reconstructPar*. La primera se encarga únicamente de la malla y está complementada por la segunda. En los casos

que involucran movimiento de malla, se deben de aplicar ambas para cada instante en el que se haya guardado datos porque para cada uno de ellos la posición de los puntos es diferente. El proceso se puede automatizar gracias a un *script* publicado por un usuario del foro de *cfd-online*. Después de ejecutarlo, dentro de la carpeta del caso ya aparecen los ficheros temporales habituales y se puede ejecutar paraFoam para observar los resultados del dominio completo -ver figura 36-.

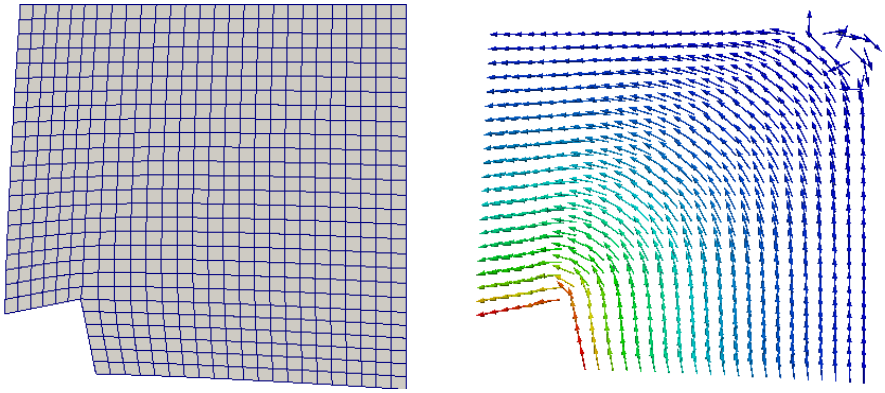


Figura 35. Movimiento de la malla y campo de velocidades en un subdominio del caso *cavity* con obstáculo en rotación.

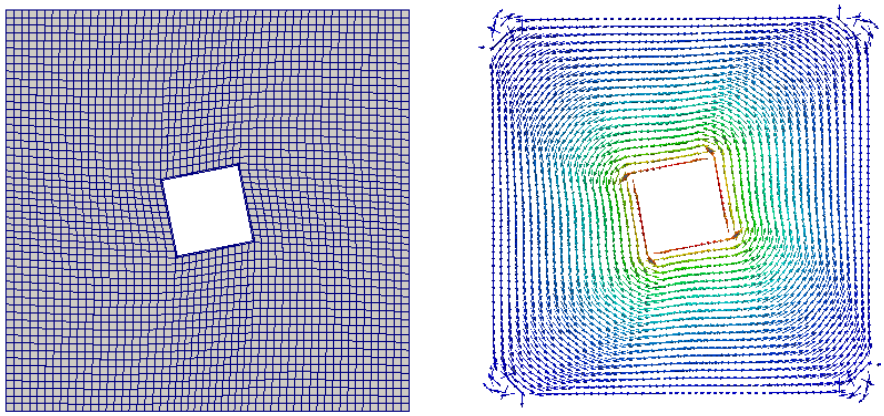


Figura 36. Movimiento de malla y campo de velocidades del dominio completo del caso *cavity* con obstáculo en rotación tras la reconstrucción de los subdominios.

La simulación realizada en paralelo es la primera parte de la rotación -parte 1- en el caso sin viento y con malla fina -2916 celdas-, que costaba 123 segundos en un único procesador -*Intel Pentium M 1.73GHz*-. Resulta interesante observar el tiempo de ejecución de este caso en función del número de procesadores utilizados. La tabla 7 incluye algunos de los datos obtenidos.

Procesadores	Tiempo [s]
1	89
2	44
4	38
6	92
8	105
10	132
14	155
18	195

Tabla 7. Tiempo de ejecución de la primera parte de la rotación del caso *cavity* con obstáculo en rotación en función del número de procesadores utilizados. Procesadores: *Intel Xeon X5355 2.66 GHz* y *Intel Xeon X3220 2.4 GHz*.

Según [1], la calidad de un proceso en paralelo con *n* procesadores se puede definir mediante una eficiencia E_n , definida de la siguiente forma:

$$E_n = T_s / n \cdot T_n$$

donde T_s es el tiempo de ejecución en un solo procesador, *n* el número de procesadores y T_n el tiempo de ejecución con *n* procesadores. El valor ideal de este parámetro es 1 -100%-, aunque habitualmente es inferior. Sin embargo, se puede llegar a superar en algunas ocasiones con 2 o 4 procesadores, lo que significa que cada uno está trabajando más rápido que si se encargara de la simulación completa por si solo. Según los datos de la tabla de arriba, la eficiencia para 2 procesadores es la más alta -101%- mientras que para el resto de casos es muy inferior. Así, aunque la simulación con 4 procesadores es la más rápida, no es la que mejor está aprovechando de la mejor forma posible la capacidad de cálculo de los ordenadores tal y como muestra su baja eficiencia -58%-.

Como se ha mencionado anteriormente, la comunicación entre procesadores retrasa la simulación hasta poderse dar el caso de sobrepasar el tiempo con un solo procesador. Se observan, en particular, un gran incremento entre los resultados de 4 a 6procesadores. La causa de este fenómeno es el hecho de que cada ordenador tiene 4 procesadores, y cuando la simulación utiliza más de 4 es necesaria la comunicación entre ordenadores. Esta transferencia de datos interna del *cluster* tiene mucho más efecto en la ejecución que la comunicación entre procesadores dentro de un mismo ordenador.

Se observa que, para el caso estudiado, el número de procesadores óptimo sería 4.

Además, se demuestra que no tiene sentido utilizar 10 o más procesadores porque el tiempo superaría al caso de uno solo.

8.9 CAMBIO DE GEOMETRÍA: EL ENGRANAJE

Una vez se ha conseguido rotar el obstáculo creado en el tutorial *cavity* ya se dispone de suficiente experiencia como para probar el giro de geometrías más complicadas. En concreto, se va a tratar de simular una geometría formada por una rueda dentada diseñada para ser incorporada en una turbo-bomba, tal y como se ha mencionado en el objetivo del estudio. En este caso la malla no sigue ningún tipo de patrón ni orden, lo que recibe el nombre de malla no estructurada. Este tipo de mallas muestra generalmente un entramado triangular a diferencia de las obtenidas mediante *blockMesh*, que siempre dan lugar a figuras cuadriláterales que siguen un cierto orden geométrico. Además, el número de puntos es significantemente mayor que el del caso *cavity* con obstáculo en rotación, hecho que va a aumentar el tiempo de cálculo ya que se debe de obtener el desplazamiento de muchos más elementos.

La rueda dentada es una malla creada en un software comercial llamado *Gambit*, que ha sido importada a OpenFOAM. Ejecutando un simple aplicación se crea la malla y ya se puede incorporar a un caso para que sea tratado su movimiento. Sin embargo, como la malla es importada no es posible crear nuevas cada cierto ángulo de rotación. Ahora no es OpenFOAM el que genera la malla, sino un software externo.

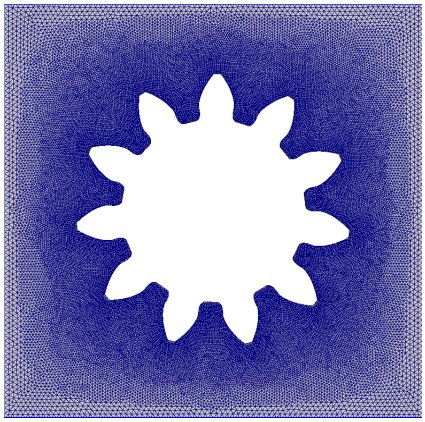


Figura 37. Malla utilizada para el análisis del engranaje.

La simulación de la rotación de la rueda dentada queda entonces limitada hasta el ángulo en el que aparezcan errores en la malla que interrumpan la simulación. Por el hecho de tratarse de movimiento automático de malla, siempre llegará un punto en el que la

geometría del entramado de los puntos será inválida.

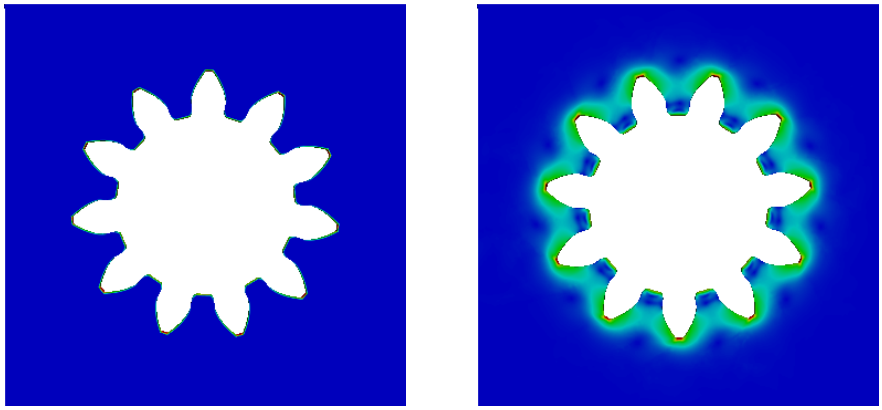


Figura 38. Campo de velocidades en la rotación del engranaje. La imagen de la izquierda muestra el instante inicial; la de la derecha al cabo de aproximadamente 15º de rotación.

Para obtener cual es el ángulo que soporta la malla, se ejecuta la simulación con un tiempo final relativamente alto en *controlDict*. De este modo se asegura que el proceso será finalizado por errores y no por criterios establecidos por el usuario.

El tiempo de ejecución es mucho mayor que en el caso de la geometría sencilla del obstáculo cuadrado. El error que finaliza el proceso ocurre a los 14700s de tiempo de ejecución, es decir, aproximadamente a las 4 horas. Por esta razón se acude a la simulación en paralelo para intentar reducirlo. Se descompone el dominio en 4 partes mediante el método *simple* y se ejecuta la simulación. Posteriormente se ejecuta también con 8 y 16 subdivisiones para comparar tiempos.

Figura 39. Diferentes formas de descomponer el dominio del engranaje. A la izquierda mediante el método *simple*; a la derecha mediante *metis*.

En la ejecución con 16 procesadores, ocurre un error. El volumen de una de las celdas del dominio se reduce demasiado y se imposibilita la continuación de la simulación

porque se produce una divergencia. La causa está relacionada con el método de descomposición -*simple*-, que no tiene en cuenta la complejidad de la malla a la hora de dividir. Por eso, se escoje un método diferente llamado *metis* que *e*s especialmente útil para mallas complejas ya que realiza la descomposición por el mejor sitio posible teniendo en cuenta su geometría. En mallas simples y estructuradas como la del caso *cavity* no hace falta realizar este paso.

Utilizando el método de descomposición *metis* se realiza la simulación con varios grupos de procesadores para ver la evolución del tiempo de ejecución. Se observa, como en el caso de obstáculo cuadrado, que la simulación más rápida tiene lugar con 4 procesadores. Sin embargo, a mayores números el tiempo no se incrementa en tan gran medida.

Si analizamos de nuevo la eficiencia propuesta por [1], se observan resultados muy similares al caso de rotación del obstáculo cuadrado. La ejecución más eficiente corresponde a la de 2 procesadores con un valor de 100.8%, frente al 80% del caso con 4 procesadores. A partir de ahí, el resto de ejecuciones presentan eficiencias muy reducidas, hasta llegar al caso de un 7% en la ejecución con 32 procesadores. Esta vez, en comparación con la simulación del cuadrado, la opción más rápida -4 procesadores- tiene una eficiencia mucho mayor.

Procesadores	**Tiempo ejecución [s]**
1	5416
2	2686
4	1685
8	1726
16	1857
32	2327

Tabla 8. Tiempo de ejecución de la rotación del engranaje hasta el error de la malla en función del número de procesadores utilizados. Procesadores: *Intel Xeon X5355 2.66 GHz* y *Intel Xeon X3220 2.4 GHz*.

La simulación se detiene aproximadamente a los 0.31s reales de rotación, lo que significa que se ha alcanzado un ángulo de rotación de 35.52º -la velocidad angular es de 2rad/s-. Afortunadamente, se observa que el ángulo de giro que aguanta la geometría sin errores es superior al que hay entre dientes: 32.72º. Esto permite poder rotar el engranaje hasta un punto en que los dientes encajan con la situación inicial. Si se para la simulación en ese instante, se puede volver a utilizar la malla original tratándola como si hubiera girado exactamente el ángulo entre dientes. Así, se puede dividir la rotación completa en tantas

partes como dientes tiene el engranaje e ir alimentando cada parte con la interpolación de los datos anteriores. Además, existe una ventaja respecto al caso del obstáculo cuadrado: no es necesario realizar el cambio de malla -bloques- cada cierto tiempo. Simplemente se empieza con la malla original, se hace rotar y se vuelve a repetir el proceso interpolando los resultados de la parte anterior.

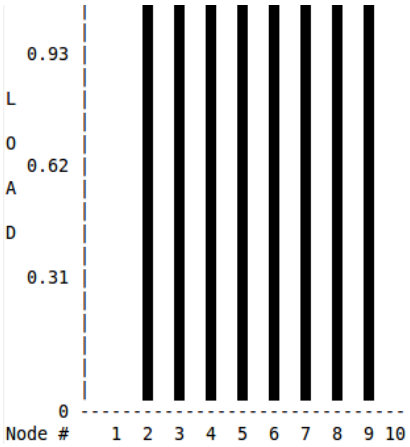


Figura 40. Situación de carga del *cluster* Mosct en un instante de la simulación del engranaje con 32 procesadores.

Interesa automatizar el proceso porque no es práctico tener que estar pendiente de la finalización de una parte de la rotación para lanzar la siguiente. Además, la duración total de la rotación supera las 4 horas y media sin tener en cuenta el tiempo de transferencia de datos de un ordenador a otro. Por lo tanto, se desarrolla un *script* que se ocupe de todo el proceso necesario:

1. Intepolar campos si no es la primera parte de la rotación
2. Descomponer el dominio
3. Eliminar la lista de coordenadas que aparecen en cada *pointMotionU* descompuesto
4. Ejecutar la simulación
5. Reconstruir malla y campos
6. Copiar/eliminar archivos por motivos de espacio
7. Volver a 1

Podría pensarse que una opción sería descomponer de antemano todos los dominios y así unicamente se debería de ejecutar e interpolar. Sin embargo, hay que recordar que la descomposición también afecta a las variables del fluido, y en todas las partes menos en la primera estas contienen la información obtenida en su antecesora.

El paso número 6 consiste en copiar fuera de Mosct cada parte de la simulación una vez finalizada y eliminarla para liberar espacio. Cada parte tiene un tamaño de aproximadamente 100MB, y teniendo en cuenta que hay 11 partes el total supera 1.1GB. Aunque el espacio disponible es de 2GB, se ha preferido llevar a cabo esta opción porque en caso de no haber suficiente memoria se perderían los datos y se tendrían que repetir las largas simulaciones. Además, hay que tener en cuenta que OpenFOAM -300MB aproximadamente- también está instalado en Mosct.

8.9.1 SCRIPT

Como en el caso de las interpolaciones con el obstáculo cuadrado, se va a mostrar una parte del *script* que se ocupa de implementar el giro completo del engranaje. El resto de partes, son tratadas con el mismo código. La única variación reside en el cambio de nombre de las carpetas a las que se accede.

La primera parte del código se ocupa de realizar la interpolación de la parte anterior. Primero se muestra un mensaje que indica en que parte de la rotación se encuentra: *GEAR9*, que hace referencia a la parte 9. A continuación indica también que se va a llevar a cabo la interpolación, que se ejecuta mediante el comando *mapFields*. Como en el caso de *cavity*, en las interpolaciones aparece un archivo no deseado -pointMotionU.unmapped- por el hecho de no incluir *pointMotionU*. Se elimina y se copia de la carpeta que contiene todas las partes del giro.

```
#------------------------------------------------------------------------#
echo
echo GEAR9
cd gear9/
echo "mapping fields from GEAR8"
mapFields ../gear8 -sourceTime latestTime > log.mapFields 2>&1
rm -rf 0/pointMotionU.unmapped
cp ../gear1/0/pointMotionU 0/
```

Código 35. Parte del *script* utilizado para la simulación de la rotación completa del engranaje.

Una vez se han inicializado los campos, ya se puede realizar la descomposición del dominio. Después, se elimina la lista de coordenadas *p0* incluidas en *pointMotionU* de cada *processor* mediante el *script* llamado *run* que debe de lanzarse desde la carpeta *processor0/0* por la forma en la que está escrito. Básicamente la misión de *run* es ir recorriendo todas las carpetas *processor* para acceder a *pointMotionU* y eliminar la lista.

```
decomposePar
gunzip processor*/0/*gz
cp ../run processor0/0
cd processor0/0
echo "Erasing p0 list from processors"
./run
cd ../../
```

Código 36. Parte del *script* utilizado para la simulación de la rotación completa del engranaje.

Una vez descompuesto el dominio empieza la simulación en paralelo mediante la orden *mpirun*. Es muy importante que al acabar la ejecución se reconstruyan los datos para que la interpolación en la parte siguiente funcione correctamente. Para poder realizar la reconstrcción sin errores se debe de eliminar el archivo *pointMotionU.gz* de cada carpeta *processor*. Por alguna razón ese campo entrabanca la reagrupación de datos y malla que se lleva a cabo mediante otro *script*, esta vez llamado *reconstrucTotal*. Este *script* se ocupa de ejecutar *reconstructParMesh* y *reconstructPar* para cada carpeta temporal guardada.

```
echo "Parallel run"
nohup mpirun -np 4 -hostfile /home/mf/usuaris/e3559475/machines icoDyMFoam
-parallel > log.icoDyMFoam 2>&1
cd ../
rm -rf processor*/0*/pointMotionU.gz
./reconstructTotal
```

Código 37. Parte del *script* utilizado para la simulación de la rotación completa del engranaje.

Finalmente, se envía el caso recién simulado fuera de Mosct y se elimina la carpeta que corresponde a la parte 7, es decir, dos partes hacia atrás. Así se mantiene espacio en Mosct y la parte 8 no se elimina hasta la siguiente parte para poder partir de ella si ha ocurrido algún error.

```
scp -r gear9/ jose@marro.upc.es:/home/jose/GEAR
rm -rf gear7/
```

Código 38. Parte del *script* utilizado para la simulación de la rotación completa del engranaje.

8.9.2 RESULTADOS

Gracias a las interpolaciones se dota a la simulación de continuidad. Esto permite que el flujo pueda adaptarse cada vez más a la rotación a medida que avanza la simulación de los 360º. Así, en las últimas partes el flujo está mucho más desarrollado que en las

primeras, donde aparece bastante circulación a muy baja velocidad en zonas lejanas a los dientes.

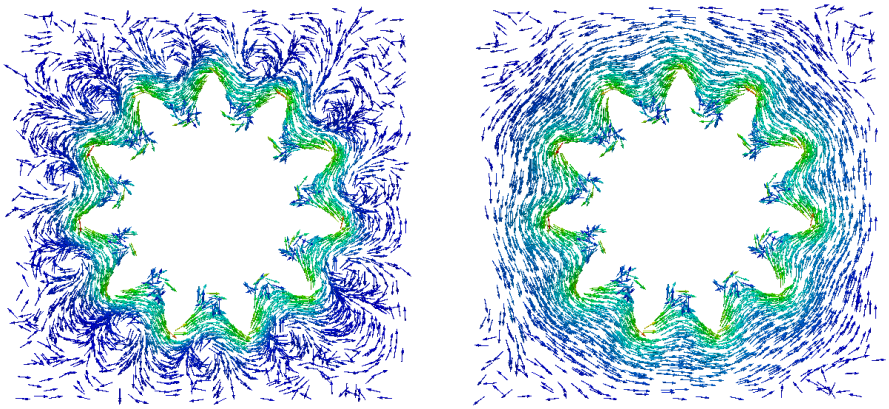


Figura 41. Campo de velocidades en la rotación del engranaje. La imagen de la izquierda muestra el instante 0.29s; la de la derecha el 2.9s.

9 DESARROLLOS FUTUROS

Para poder llevar a cabo la simulación completa de la bomba oleohidráulica investigada en la ETSEIAT -[13]- queda aún cierto camino por recorrer. En primer lugar es necesario que se utilice un mecanismo propio de OpenFOAM para generar mallas como la importada de *Gambit*. Es más interesante poder preprocesar de forma totalmente independiente del software comercial *Fluent*. La aplicación más adecuada es, a priori, *snappyHexMesh*. Se puede generar la imagen 3D con cualquier tipo de programa de diseño por ordenador y después aplicarla a OpenFOAM con las debidas modificaciones.

El hecho de incluir dos engranajes complica el tratamiento desde el punto de vista de la malla dinámica. Después de entrar en contacto, los dientes se separaran unos de otros rápidamente y ,por lo tanto, se requerrirán periodos de rotación entre regeneraciones de malla más pequeños para mantener su calidad. Además, no se puede aprovechar la misma malla de partida para cada una de las partes de la rotación tal y como sucedia con un solo engranaje como se ha mostrado anteriormente. Se deben de crear mallas intermedias de la misma forma que se ha desarrollado en el estudio para el caso del obstáculo cuadrado pero esta vez con una geometría bastante más compleja.

En resumen, se deben generar las diferentes mallas correspondientes a cada una de las partes de la rotación que deben de ser lo suficientemente pequeñas para mantener la calidad de los resultados. Cabe mencionar que no hace falta que se generen mallas para los 360º de giro por la periodicidad del movimiento. Es suficiente alcanzar el punto en el que los dientes se han desplazado hasta el lugar en el que se encontraba el diente precedente al principio de la simulación. A partir de ahí el comportamiento de la malla se puede repetir hasta llegar al límite deseado.

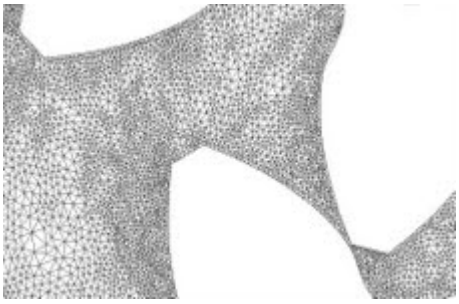


Figura 42. Vista de varios dientes de los dos engranajes de la bomba oleohidráulica investigada en la ETSEIAT. Se muestra la zona de encaje. Malla creada con *Gambit*. Fuente: [13].

Si se consigue tratar con éxito la deformación de la malla, el *solver* incompresible *icoDyMFoam* se puede ejecutar, en principio, sin ninguna modificación adicional excepto las correspondientes variaciones de las condiciones de contorno del nuevo dominio.

10 CONCLUSIONES

A lo largo del estudio se han desarrolado varias pasos intermedios con el objetivo de evolucionar escaladamente hasta llegar al objetivo: conseguir implementar desde el punto de vista de la malla el giro de un engranaje.

En primer lugar se ha modificado la malla de un caso sencillo añadiendo un obstáculo en el centro del dominio. A continuación, para poder desplazarlo y que las simulaciones se lleven a cabo de forma rápida, se ha desarrollado un *solver* incompresible capaz de tratar mallas dinámicas. Posteriormente, se ha implementado -en el obstáculo creado- el movimiento de translación en 1 y 2 dimensiones y se ha creado una nueva condición de frontera necesaria para aplicar rotaciones con velocidad angular constante. Se ha aplicado giro al obstáculo junto con interpolaciones de datos y generación de nuevas mallas para abarcar giros de grandes ángulos, todo ello de forma automatizada mediante el uso de *scripts*. Por último, el obstáculo se ha substituido por un engranaje y se ha simulado su rotación mediante el uso de múltiples procesadores, consiguiendo abordar la finalidad del estudio con éxito.

Se ha observado que de las formas de crear mallas con OpenFOAM, la más sencilla es mediante el uso de *blockMesh* siempre que se trate de geometrías simples. Cuando el dominio que se pretende representar es complejo la alternativa más adecuada es la importación, si se dispone de un software para crearla, o la creación a partir de una imagen tridimensional en formato STL: *snappyHexMesh*.

Respecto al tratamiento de la deformación de malla se puede concluir que el efecto de la difusividad tiene una repercusión decisiva. Es muy importante elegir correctamente el esquema en función del tipo de desplazamiento que se pretenda simular. Cada una de las opciones disponibles tiene su correspondiente aplicación más adecuada y si se usa de forma incorrecta el comportamiento de los puntos se puede ver afectado negativamente.

Por otro lado, la elección de esquema de difusividad es también relevante desde el punto de vista computacional, sobretodo si hay fuertes restricciones de tiempo. Cuanto más complejo es el modelo, más esfuerzo de cálculo se necesita y, por lo tanto, el tiempo de la simulación se alarga. Este aspecto hace que en algunos casos la elección del tipo de difusividad se obtenga a partir de un compromiso entre calidad de malla y tiempo de ejecución.

El movimiento automático de los puntos internos de la malla no soporta grandes

deformaciones. Las celdas no pueden adoptar cualquier forma y en caso de verse excesivamente forzadas generan errores que no permiten continuar con los cálculos. Ante esta situación la principal alternativa es el uso de cambios topológicos, es decir, la introducción o substracción de celdas y/o la modificación de su conectividad con sus vecinas. En OpenFOAM, dichos cambios se suelen agrupar para formar códigos que se activan automáticamente durante la simulación en función de ciertos criterios establecidos por el usuario. Sin embargo, las relativamente limitadas opciones -en OpenFOAM- obligan a implementar nuevas funcionalidades para tratar casos con particulariedades aún no soportadas en la distribución oficial.

El uso de *scripts* es particularmente práctico en OpenFOAM por el hecho de que las aplicaciones se ejecutan desde el terminal de comandos. Mediante códigos relativamente cortos se pueden automatizar ciertos procesos que conllevarían mucho tiempo si se realizaran a mano. En el estudio se han utilizado por ejemplo para la automatización de la simulación de rotaciones por partes.

En cuanto a la descomposición de dominios para la ejecución en paralelo, los métodos utilizados han sido *simple* y *metis*. Se ha observado la idoneidad del primero en la aplicación a dominios de geometría simple. Su función es básicamente dividir el dominio en partes iguales en cada una de las direcciones que haya seleccionado el usuario. En cambio, *metis* realiza la división de forma óptima en función de la configuración de la malla en el momento de la descomposición. El resultado son varios subdominios de diferente tamaño con una mayor robustez frente a deformaciones. Por lo tanto, es mucho más adecuado cuando se tratan mallas complejas y, sobretodo, si éstas van a tener que soportar una cierta deformación.

Según la experiencia que se ha tenido durante el estudio en ejecuciones en paralelo se puede concluir que lo ideal es tener el máximo número de procesadores dentro de un mismo ordenador. En cuanto el proceso involucra comunicación entre varios ordenadores, aparece un importante aumento de tiempo que reduce drásticamente la eficiencia de la ejecución. De todos modos, no se debe utilizar el máximo número de procesadores aunque estén dentro de un mismo ordenador por el hecho de existir también comunicación entre ellos. Habitualmente existe un número óptimo de procesadores que dan lugar al mínimo tiempo de simulación y a partir de ahí, si se utilizan más el tiempo siempre será mayor.

Refiriendo de nuevo a los tiempos de ejecución, se ha observado que cuanto más larga es la simulación en un solo procesador, mejor será su comportamiento en un proceso en paralelo. Simulaciones cortas con escasos puntos de cálculo funcionan bien con pocos

procesadores -2 o 4-, pero si el número aumenta es fácil que la simulación sea más larga que cuando se aplica a un solo procesador. En cambio, cuando se trata de simulaciones de mallas relativamente densas con tiempos de ejecución altos, se puede utilizar un número más amplio de procesadores sin que el tiempo aumente de forma tan brusca como en los casos sencillos.

11 PRESUPUESTO

Según los créditos que posee un Proyecto de Final de Carrera en la Escuela Técnica Superior de Ingenierías Industrial y Aeronáutica de Terrassa de la Universidad Politécnica de Catalunya -UPC- las horas de trabajo correspondientes son 375. Suponiendo que el sueldo de un ingeniero en prácticas es de 8€/hora el coste personal asciende a 3000€.

Para la realización del estudio ha sido necesario cierto material informático. En concreto se han utilizado un ordenador personal, un ordenador fijo vinculado a la UPC y el *cluster* de cálculo de los servicios informáticos del Campus Terrassa UPC. El coste del ordenador personal es 1000€, el del vinculado a la UPC X y el del *cluster* 31500€.

Finalmente, se debe de tener en cuenta el coste del software empleado. Ha sido necesaria una instalación del sistema operativo Linux y de OpenFOAM, ambos de libre distribución. Además, se ha recurrido al uso del software comercial Fluent, en concreto su preprocesador Gambit, para generar la malla del engranaje cuya licencia educativa es de 300€.

Concepto	**Coste [€]**
Horas de trabajo	3000
Cluster	31500
Ordenador UPC	800
Ordenador propio	1000
Licencia educativa Fluent	300
TOTAL	36600

Tabla 9. Presupuesto desglosado de los recursos necesarios en el desarrollo del estudio.

12 IMPLICACIONES MEDIOAMBIENTALES

En este PFC no se ha tenido en cuenta ningún tipo de factor medioambiental, puesto que ha sido realizado con ordenador. En consecuencia, se considera que no ha habido ninguna implicación medioambiental fuera de la normativa vigente.

13 BIBLIOGRAFÍA

[1] Ferziger, J.H., Perić, M. (2002). *Computational Methods for Fluid Dynamics*. Germany: Springer

[2] Jasak, H., Tuković, Ž. (2006). *Automatic Mesh Motion for the Unstructured Finite Volume Method* , Transactions of FAMENA, Vol. 30, No. 2, 2007, pp. 1–18

[3] Liefvendahl, M., Troëng, C. (2007). *Deformation and Regeneration of the Computational Grid for CFD with Moving Boundaries*. 45th AIAA Aerospace Sciences Meeting and Exhibit , January 2007, Reno, Nevada

[4] Moradina, P. (2008). *Project work for the PhD course in OpenFOAM*. Lund University, Göteborg, Sweden

[5] Bos, F., van Oudheusden, B., Bijl, H. (2008). *Moving and deforming meshes for flapping flight at low Reynolds number*, OpenFOAM Workshop, Milan, july 2008

[6] Jasak, H. (2009). *Dynamic Mesh Handling in OpenFOAM*, 47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition, Orlando, Florida, january, 2009

[7] Jasak, H., Rusche, H. (2009). *Dynamic Mesh Handling in OpenFOAM*, 4th OpenFOAM Workshop, Montreal, Canada, june 2009

[8] Nilsson, H., Petit, O. (2009). *Pre-processing in OpenFOAM. Mesh generation.* OpenFOAM course 2009. Chalmers University of Technolgy.

[9] Berce, A. (2010). *OpenFOAM tutorial. Adaptive mesh refinement*. Chalmers University of Technology.

[10] Lucchini, T. (-). *OpenFOAM programming tutorial*. Department of Energy, Politecnico di Milano.

[11] OpenCFD Limited. (2010). *OpenFOAM User Guide*. Version 1.7.1, August 2010.

[12] OpenCFD Limited. (2010). *OpenFOAM Programmer's Guide*. Version 1.7.1,

August 2010.

[13] Del Campo, D. (2011). *Analysis of the Suction Chamber of External Gear-Pumps and their influence on Cavitation and Volumetric Efficiency*. Tésis doctoral.

[14] Cfd-Online OpenFOAM Forum. Consulta: octubre 2010 - mayo 2011
<http://www.cfd-online.com/Forums/openfoam-code/>

[15] OpenFOAM unofficial Wiki. Consulta: diciembre 2010 - enero 2011
<http://openfoamwiki.net>

[16] OpenFOAM 1.7.x Documentation. Consulta: diciembre 2010 - mayo 2011
<http://www.openfoam.com/docs/cpp/>

[17] Repositories OpenFOAM-extend. Consulta: diciembre 2010 - enero 2011
<http://openfoam-extend.svn.sourceforge.net/viewvc/openfoam-extend/>

[18] OpenFOAM Terrassa googleGroups. Consulta: setiembre 2010 - febrero 2011
<http://groups.google.com/group/openfoam-terrassa>

[19] OpenFOAM website. Consulta: setiembre 2010 - mayo 2011
<http://www.openfoam.com>

[20] Ubuntu Forums. Consulta: setiembre 2010 - mayo 2011
<http://ubuntuforums.org/>

[21] Linux Questions. Consulta: setiembre 2010 - mayo 2011
<http://www.linuxquestions.org/>

[22] C++ Language Tutorial. Consulta: setiembre 2010 - enero 2011
<http://www.cplusplus.com/doc/tutorial/>

[23] Linux Shell Scripting Tutorial. Consulta: diciembre 2010 - enero 2011
<http://www.freeos.com/guides/lsst/index.html>